Linux 内核修炼之道精华版

之

方法论

任桥伟

blog.csdn.net/fudan_abc

目录

前言 3
精华版方法论部分导读 **5**
LINUX 大史记 5
内核学习的方法论 5
驱动开发的方法论 7
LINUX 内核问题门 7
缅怀已逝的十八年（1991～1998） **8**
LINUX 诞生记 8
泰坦尼克的狂潮 9
缅怀已逝的十八年（1999～2002） **10**
提前发生的革命 10
和平、爱情和 LINUX 11
缅怀已逝的十八年（2003～2006） **11**
UBUNTU 4.10 11
RICHARD STALLMAN 的征婚启事 12
缅怀已逝的十八年（2007～2009） **14**
来自微软的指控 14
首款 ANDROID 手机 17
LINUX 信用卡 17
KERNEL 地图：KCONFIG 与 MAKEFILE **18**
MAKEFILE 不是 MAKE LOVE 18
利用 KCONFIG 和 MAKEFILE 寻找目标代码 19
分析内核源码如何入手？（上） **21**
分析 README 21
分析 KCONFIG 和 MAKEFILE 23
分析内核源码如何入手？（下） **26**
态度决定一切：从初始化函数开始 26
内核学习的心理问题 **31**
内核学习的相关资源 **32**
内核文档 32
经典书籍 33
内核社区 34
其他网络资源 34
模块机制与“HELLO WORLD!” **35**
设备模型（上） **37**
设备模型（下） **42**
内核中 USB 子系统的结构 42
USB 子系统与设备模型 43
驱动开发三件宝：SPEC、DATASHEET 与内核源码 **44**
LINUX 内核问题门——学习问题、经验集锦 **45**
LINUX 内核学习常见问题 45
LINUX 内核学习经验 47

前言

至此落笔之际，恰至 Linux 问世 18 周年，18 年的成长，如梦似幻，风雨颇多，感慨颇多。

犹自忆起多年以前一位前辈训导时的箴言：今天的必然正是由之前一系列的偶然所决定的。过去的某年某月，我偶然初识 Linux 就身陷其中，至今仍找不到出去的路，而正是这次乃至之后的多次偶然相联合，从而决定了今日的我要在此写下这些话。那么，当您偶然地拿起这本书，偶然地看到这段话，您是否会问自己：这样的偶然又会导致什么样的必然？

如果您依然决定继续这次的偶然之旅，那么首先请认识两个人，准确的说是一个人和一只企鹅。这个人自然就是 Linus Torvalds，我们也可是称他为 Linus 或李纳斯，正是这位来自芬兰的天才，在 1991 年 1 月 2 日，攥着在圣诞节和生日得到的钱，偶然地做出了一个重大的财政决定，分期三年买一台价格 3500 美元得相貌平平得计算机，从而 Linux 开始了。

企鹅则是 Linux 的标志，很多人可能不知道 Linus，但是却可能知道这只企鹅，这是一个奇怪的现象，就像很多人知道微软，却不知道比尔盖茨。不管怎么说，是 Linus 塑造了这只企鹅，并让它有一副爽透了的样子，就像刚刚吞下一扎啤酒。除此之外，这只企鹅还要很特别，其他的企鹅都是黑嘴巴黑脚蹼，但它却是黄嘴巴黄脚蹼，这使它看上去好像是鸭子与企鹅的杂交品种，也许它是唐老鸭在南极之旅中与一只当地企鹅一夜倾情的结晶。

其次，在您继续之前，我还想请您问自己一个问题：我在强迫自己学习内核么？我很希望您能回答不是，但希望与现实往往都有段不小的距离，因为很多时候，我都发现身边的人是因为觉得内核很高深而强迫自己喜欢的。强迫自己去喜欢一个人是多么痛苦的事情。或许，针对这个问题，最让人愉悦的回答是“说实话，我学习的热情从来都没有低落过。”正如 Linus 在自己的自传《Just for Fun》中希望的那样。

本书的组织形式

本书将 Linux 内核的学习分为四个层次：全面了解，掌握基本功；兴趣导向，选择重点深度钻研；融入社区，参与开发做贡献；坚持，坚持，再坚持。总结起来，就是“全面了解抓基本，兴趣导向深钻研；融入社区做贡献，坚持坚持再坚持。”（如果您是一个修真小说爱好者，尽可以将其与炼气、筑基、结丹和元婴等层次相对应。）

第一层次修炼的内容包括了前三章，目的是希望您能够对 Linux 以及内核有个全面的认识和了解，掌握分析 Linux 内核源代码的分析方法。

第 1 章主要介绍了 Linux 的 18 年成长史，或许您会乐意陪我一起缅怀下这过去的十八年。

第 2 章介绍内核的配置和编译过程，和任何大型软件源码的学习一样，学会编译和配置是第一步。

第 3 章介绍学习内核需要的基础，内核的体系结构、目录结构、代码特点，浏览内核代码的工具，最后，突出强调了内核源码分析过程中极为重要的两个角色——Kconfig 和 Makefile，并以 USB 子系统为例，演示了如何利用这两个角色进行代码分析。

第二层次的修炼包括了第 4～11 章的内容，对内核多数部分的工作原理进行介绍。按照

认识的发展规律，在第一层次修炼中已经对内核有个全局的认识和了解之后，接下来就应该以兴趣为导向，寻找一个子系统或模块，对其代码深入钻研和分析，不懂的地方就通过社区、邮件列表或者直接发 Email 给 maintainer 请教等途径弄懂，切勿得过且过，这样分析下来，对同步、中断等等内核的很多机制也同样会非常了解，俗话说一通则百通就是这个道理。

因此第二层次的各个章节里，只是阐释重点的概念和工作原理，帮助您在分析该部分代码时进行理解，并不求详尽。

第 4 章讨论系统的初始化，万事开头难，系统的初始化是一个很复杂的过程，不过对于内核源码的学习来说，以这个部分开始应该是个不错的选择。特别是子系统初始化的讨论，应该是您选择任何内核子系统开始分析时都需要了解的内容。

第 5 章讨论系统调用，它是应用程序和内核间的桥梁，学习并理解它是我们走向内核的一个很好的过渡。

第 6 章讨论内核的中断处理机制，包括几乎任何一本内核书籍都没有涉及的通用 IRQ 层。

第 7 章讨论进程的内核抽象，以及进程如何被创建和销毁。如果我们将计算机上运行的操作系统以及各种各样的软件看作一系列有机的生命体，而不是死的指令集合，那么这就是一个进程的世界，只不过与我们人类世界不同的是，进程世界里的个体是一个一个鲜活的进程，而不是人。人的世界有道德与法律去制约管理，进程的世界同样也有自己的管理机制，这就是第 7 章所要展示的内容——进程管理。

第 8 章讨论进程的调度，重点讨论了在内核历史上具有重要地位的 O(1)调度器和最新的 CFS 调度起。

第 9 章讨论内存管理，内存就是进程的家，这里讨论内核如何为每个进程都分配一个家，并尽量的去做到“居者有其屋”，以及保证每个家的安全。

第 10 章讨论文件系统，主要是虚拟文件系统（VFS），它通过在各种具体的文件系统之上建立一个抽象层，屏蔽了不同文件系统间的差异。

第 11 章讨论设备驱动，对于驱动开发来说，设备模型的理解是根本，spec、datasheet 与内核源代码的利用是关键。

通过第二层次的修炼，您应该对至少一到两个部分有了很深入的理解，对内核代码采用的通用手法也已经很拈熟，那么您应该开始进入第三层次，努力融入到内核的开发社区，此时的您已经不会再是社区中潜水小白的角色，而是会针对某个问题发表自己的见解。您已经可以尝试参与到内核的开发中去，即使仅仅修改了内核中的一个错误单词，翻译了一份大家需要的文档，也是做出了自己的贡献，会得到大家的认可。

本书中第三层次只包括了两章的内容，这是因为内核的修炼之道越往后便越依赖于自己，任何参考书都替代不了自己不断的反思与总结。

第 12 章讨论参与内核开发需要了解的一些基础信息。

第 13 章讨论了内核的调试技术，与第 12 章一样，您可以仅仅将这些内容看成内核修炼中的一些 tips。

至于最后的第四层次，更是仅有两个字——坚持。能够在内核的修炼之道上走多远，都

取决于我们能够坚持多久，或许一个用一个公式概括更为合适：心态+兴趣+激情+时间+X＝Y。

革命尚未成功，我等仍需努力。——与君共勉之。

精华版方法论部分导读

到目前为之，博客上分享的精华篇都可以归为方法论的范畴，在很多时候，都是方法论要比细节紧要得多。而这些精华篇又可细分为三个专题：Linux 大史记；内核学习的方法论；驱动开发的方法论。

Linux 大史记

除去那些精彩的“门”，我们生活中乏味的事情太多了，所以不希望再去按惯例花个一二页的篇幅乏味的写个“Linux 简介”，就将几天中出去溜弯的时间贡献了出来，逐年逐月的搜集整理了一些 Linux 成长过程中所发生的重要的事情，抑或一些非常有趣儿的事情。

开始时本以为这是一件很轻易的事，起码应该比统计公布房价上涨多少的事情轻易的多，利用 google，完成这么一件事情又有何难？但是意外的是，貌似很难找到类似的归纳整理，或许能够看到某个时间段内的所谓的 top10 之类的字眼，但里面的罗列似乎大都满足不了有趣儿的要求。所以里面有些月份是个空白，不管如何，大家可以了解了解，看看是否有很多自己不知道的有趣闻轶事？

缅怀已逝的十八年（1991～1998）

缅怀已逝的十八年（1999～2002）

缅怀已逝的十八年（2003～2006）

缅怀已逝的十八年（2007～2009）

内核学习的方法论

透过现象看本质，兽兽门无非就是一些人体艺术展示。同样往本质里看过去，学习内核，就是学习内核的源代码，任何内核有关的书籍都是基于内核，而又不高于内核的。

所以这个专题的前三个精华篇就是专注于介绍如何入手分析内核源代码的，这里前无来者的突出强调了“Kernel 地图”的概念，虽然 Goggle 带着 Goggle 地图远去了，可 Kernel 地图仍然在继续。

Kernel 地图：Kconfig 与 Makefile

毫不夸张地说，Kconfig 和 Makefile 是我们浏览内核代码时最为依仗的两个文件。基本上，Linux 内核中每一个目录下边都会有一个 Kconfig 文件和一个 Makefile 文件。对于一个希望能够在 Linux 内核的汪洋代码里看到一丝曙光的人来说，将它们放在怎么重要的地位都不过分。

我们去香港，通过海关的时候，总会有免费的地图和各种指南拿，有了它们在手里我们才不至于无头苍蝇般迷惘的行走在陌生的街道上。即使在内地出去旅游的时候一般来说也总是会首先找份地图，当然了，这时就是要去买了，拿是拿不到的，不同的地方

有不同的特色， 只不过有的特色是服务，有的特色是索取。

Kconfig 和 Makefile 就是 Linux Kernel 迷宫里的地图。地图引导我们去认识一个城市，而 Kconfig 和 Makefile 则可以让我们了解一个 Kernel 目录下面的结构。我们每次浏览 kernel 寻找属于自己的那一段代码时，都应该首先看看目录下的这两个文件。

分析内核源码如何入手？（上）

既然要学习内核源码，就要经常对内核代码进行分析，而内核代码千千万，还前仆后继的不断往里加，这就让大部分人都有种雾里看花花不见的无助感。不过不要怕，孔老夫子早就留给我们了应对之策：敏于事而慎于言，就有道而正焉，可谓好学也已。这就是说，做事要踏实才是好学生好同志，要遵循严谨的态度，去理解每一段代码的实现，多问多想多记。如果抱着走马观花，得过且过的态度，结果极有可能就是一边看一边丢，没有多大的收获。

分析内核源码如何入手？（下）

下面的分析，米卢教练说了，内容不重要，重要的是态度。就像韩局长对待日记的态度那样，严谨而细致。

只要你使用这样的态度开始分析内核，那么无论你选择内核的哪个部分作为切入点，比如 USB，比如进程管理，在花费相对不算很多的时间之后，你就会发现你对内核的理解会上升到另外一个高度，一个抱着情景分析，抱着 0.1 内核完全注释，抱着各种各样的内核书籍翻来覆去的看很多遍又忘很多遍都无法达到的高度。请相信我！

让我们在 Linux 社区里发出号召：学习内核源码，从学习韩局长开始！

对于学习来说，无论是在学校的课堂学习，还是这里说的内核学习，效果好或者坏，最主要取决于两个方面——方法论和心理。注意，我无视了智商的差异，这玩意儿玄之又玄，岔开了说，属于迷信的范畴。

因此继介绍分析内核源码方法的三个精华篇之后，又针对内核学习过程中最为常见的两个心理误区做了阐述。

内核学习的心理问题

而心理上的问题主要有两个，一个是盲目，就是在能够熟练适用 Linux 之前，对 Linux 为何物还说不出个道道来，就迫不及待的盲目的去研究内核的源代码。这一部分人会觉得既然是学习内核，那么耗费时间在熟悉 Linux 的基本操作上纯粹是浪费宝贵的时间和感情。不过这样虽然很有韩峰同志的热情和干劲儿，但明显走入了一种心理误区。重述 Linus 的那句话：要先会使用它。

第二个就是恐惧。人类进化这么多年，面对复杂的物体和事情还是总会有天生的惧怕感，体现在内核学习上面就是：那么庞大复杂的内核代码，让人面对起来该情何以堪啊！

即使有好的方法和坚强的心理，我们在内核学习过程中仍需要利用很多好的资源。其实，韩峰同志已经在日记里告诉了我们资源的重要性，因此我们在学习韩峰同志严谨细致的态度同时，还要领悟他对资源的灵活运用。只有在以内核源码为中心，坚持各种学习资源的长期建设不动摇，才能达到韩局长那样的高度，俯视 Linux 内核世界里的人生百态。

内核学习的相关资源

待到山花烂漫时，还是那些经典在微笑。

驱动开发的方法论

因为至少在国内大部分内核相关的开发都是驱动的开发，所以在内核学习的方法论之后，专门用一个专题，从模块机制、设备模型、驱动三件宝三个层次介绍了驱动开发的方法论。

模块机制与“Hello World!”

有一种感动，叫泪流满面，有一种机制，叫模块机制。显然，这种模块机制给那些 Linux 的发烧友们带来了方便，因为模块机制意味着人们可以把庞大的 Linux 内核划分为许许多多个小的模块。对于编写设备驱动程序的开发者来说，从此以后他们可以编写设备驱动程序却不需要把她编译进内核，不用 reboot 机器，她只是一个模块，当你需要她的时候，你可以把她抱入怀中（insmod），当你不再需要她的时候，你可以把她一脚踢开（rmmod）。

设备模型（上）

设备模型（下）

对于驱动开发来说，设备模型的理解是根本，毫不夸张得说，理解了设备模型，再去看那些五花八门的驱动程序，你会发现自己站在了另一个高度，从而有了一种俯视的感觉，就像凤姐俯视知音和故事会，韩峰同志俯视女下属。

顾名而思义就知道设备模型是关于设备的模型，既不是任小强们的房模，也不是张导的炮模。对咱们写驱动的和不写驱动的人来说，设备的概念就是总线和与其相连的各种设备了。电脑城的 IT 工作者都会知道设备是通过总线连到计算机上的，而且还需要对应的驱动才能用，可是总线是如何发现设备的，设备又是如何和驱动对应起来的，它们经过怎样的艰辛才找到命里注定的那个他，它们的关系如何，白头偕老型的还是朝三暮四型的，这些问题就不是他们关心的了，而是咱们需要关心的。在房市股市千锤百炼的咱们还能够惊喜的发现，这些疑问的中心思想中心词汇就是总线、设备和驱动，没错，它们就是咱们这里要聊的 Linux 设备模型的名角。

驱动开发三件宝：spec、datasheet 与内核源码

设备模型之外，对于驱动程序的开发者来说，有三样东西是不可缺少的：第一是协议或标准的 spec，也就是规范，比如 usb 协议规范；第二是硬件的 datasheet，即你的驱动要支持的硬件的手册；第三就是内核里类似驱动的源代码，比如你要写触摸屏驱动的话，就可以参考内核里已经有的一些触摸屏驱动。

Linux 内核问题门

继前面三个专题之后，为了感谢精华篇发布过程中很多朋友的关心与支持，便以“问题门”为题为拙作《Linux 内核修炼之道》制作了一个小插曲,希望通过对大家内核学习过程中遇到的问题与经验心得做一番展示，来帮助还在门外的朋友寻找到这扇门的钥匙。

Linux 内核问题门——学习问题、经验集锦（持续更新中……）

陈宪章说：“学贵有疑，小疑则小进，大疑则大进。疑者，觉悟之机也，一番觉悟

一番长进。”

培根说：“多问的人将多得。”

还在学校的时候导师在激情讲演之后对着会议室里形态各异但均静默不语的我们痛心疾首的说：“会提问题很重要啊，同志们！不会提问题怎么有资格做研究！”

这样铿锵有力的训诫今日想起仍觉深受刺激，于是就要不可避免得要做出一些反应来。不过一是因为咱这年代还没有非主流的说法，二是因为也没有冯仰妍同学的性别优势，不可能受到刺激就整出个门来。咱能够做到的最大反应也就是在这里开贴专门探讨探讨内核学习的相关问题，为了稍微增加那么一些广告效应，就称为“问题门”吧。

使用“问题门”的称呼，一是内心里潜藏的那点低级趣味想去沾点近些年层出不穷各种各样的“门”的仙气，二是在内核的学习过程中的确实实在在的存在着这样的一个“门”，横亘在我们的面前，跨过去便海阔天空是另一番世界，但却是让无数人竞折腰，百思不得其钥匙。

缅怀已逝的十八年（1991～1998）

至此落笔之际，恰至 Linux 问世 18 周年，18 年的成长，风雨颇多，感慨颇多，谨以这些许年来的点滴之事为 Linux 的成人礼添彩。

如果你尚未与 Linux 亲密接触过，那么希望这里的内容可以成为你初识 Linux 的见证。如果你已经是个 Linux 达人，那么就选个安静的早晨，抑或下午，陪我一起缅怀下这过去的十八年吧。

Linux 诞生记

1987 年

MINIX 诞生，而我也已端坐于学堂之中，隐去一身的稚气，能够摇头晃脑的吟诵几句诗赋了。若真是冥冥中自有定数的话，或许这时就暗定了 4 年后 Linux 的诞生。

1991 年

Linus Torvalds，一个芬兰的大学生，对于他不能按照意愿访问大学 UNIX 服务器而感到很愤怒，于是开始为一个以后被称为“Linux”的内核而工作，并于这一年的 10 月 5 日发布了 Linux 0.01。

1992 年

4 月，第一个 Linux 新闻组“comp.os.linux”建立。10 月，第一个可以安装的 Linux 版本 SLS 发布。同年，我拿到了平生的第一个毕业证。

1993 年

8 月，第一本关于 Linux 的著作《Linux Installation and Getting Started Version 1》出版。而这一年，我最敬佩的语文老师患病离去了，从此，我知道了生活中不仅仅只有欢聚，还有伤别。

1994 年

Linux 1.0 发布，并采用 GPL（GNU General Public License，通用公共许可证）协议。大家要 Linus Torvalds 想一只吉祥物，Linus 突然想到小时候去动物园被一只企鹅追着满地打滚，还被咬了一口！既然想不到其它的吉祥物了，干脆就以这支企鹅来当吉祥物算了！

泰坦尼克的狂潮

1995 年

4 月，召开首届 Linux 博览会，一个以 Linux 为特征的商业展览博览会。几个月后，我迎来了第二个中学阶段。

1996 年

Linux 2.0 发布，它第一个支持了 SMP（对称多处理器）架构。此时 Linux 的全球用户已经达到了 350 万左右。

1997 年

首例 Linux 病毒“Bliss”被发现。电影《泰坦尼克号》所用的 160 台 Alpha 图形工作站中，有 105 台采用了 Linux。

1998 年

1 月，第一份 Linux 新闻周刊出版，同时，Netscape 宣布他们将在自由软件许可协议下发布浏览器的源代码，这为 Linux 和自由软件的发展提供了广阔空间。

2 月，Eric Raymond 和他的朋友门提出了“open source”的概念，申请了该商标特权并且组建了 opensource.org 网站，从而开始推动 Linux 的商业化发展。

4 月，Linux 广泛被美国国家公共新闻广播报道，标志 Linux 在主流、非技术性的媒体界首次出现。

5 月，Google 搜索引擎开始流行，不仅仅是因为它是最好的搜索引擎，而且还因为它是基于 Linux 和具有 Linux 特色的搜索网页。

6 月，“从来没有一个用户向我提起 Linux，Linux 就像众多的免费产品一样，虽然它是很小的，却得到了一群忠诚的拥护者。”比尔盖茨在 6 月 25 日的《PC 周刊》上说。

7 月，KDE 和 GNOME 的桌面之争在其拥护者之间愈演愈烈，Linus 以实际行动中表明 KDE 非常好用，在这种情况下，KDE1.0 诞生了。Oracle、Informix、Sybase 都宣布将积极支持 Linux。Linux 开始成为一个家喻户晓的词。

9 月，Dave Whitinge 和 Dwight Johnson 创建了 LinuxToday.com，该网站后来被 Internet.com 收购，不过它一直是访问量最高和最容易阅读的 Linux 入门网站。

12 月，一篇来自 IDC 的报导说 Linux 的发行量在 1998 年涨了 200%以上，它的市场占有率也增加了 150%以上。Linux 拥有 17%的市场占有率并且增长率超过了市场上其它任何一个系统。

同年，我迎来了人生中一个非常重要的时刻：我上大学了！

缅怀已逝的十八年（1999～2002）

提前发生的革命

1999 年

1 月，“Linux 2.2 已经发布，我终于可以松口气了”创造者 Linus Torvalds 说。

3 月，首届 LinuxWorld 讨论会和博览会在加洲的圣何塞举行，作为 Linux 第一个大的商业化的贸易展示活动，它无疑向世界昭示了 Linux 的到来。

8 月，SG 宣布了与 Red Hat 的合作关系，并且开始大规模的为内核的发展做贡献。Red Hat 进行了首次公开募股，股价马上涨到了 50 美元，在那个时候这个价似乎很高。摩托罗拉公司与 Lineo 建立了合作关系，进入 Linux 领域并提供嵌入式系统产品，支持和培训服务。Sun 宣布了在 Sun 公共源许可（Sun Community Source License）下发行 StarOffice 和开发一个网络版本的办公套件。

9 月，Red Hat 的股票达到了 135 美元，这个价格在那个时候似乎是难以置信的高。

10 月，Sun 宣布它将在 Sun 公共源许可下公布 Solaris 的源代码。

12 月，VA Linux Systems 的首次公开募股价格是 30 美元/股，这个价格很快涨到了 300 美元，它在 NASDAQ 历史上创造了最高的首次公开募股价格。

这一年，网络进入了宿舍，QQ、mud 等也进入了我们的生活。

2000 年

1 月，VA Linux Systems 宣布创建我们非常熟悉的 SourceForge，到这年底，SourceForge 已经接到了超过 12000 个项目，拥有 92000 个注册的开发者。

2 月，最近的 IDC 报告显示 Linux 现在成为“服务器电脑上第二个最受欢迎的操作系统”，在 1999 年占了 25%的服务器操作系统销售额，Windows NT 为 38%，占第一位，NetWare 为 19%，排名第三，IDC 以前曾预测过 Linux 将在 2002 或 2003 年到达第 2 位，这场革命提前发生了。

3 月，嵌入式 Linux 协会（Embedded Linux Consortium）成立。

8 月，HP、Intel、IBM 以及 NEC 宣布开放源代码发展实验室（OSDL，Open Source Development Lab）成立。

9 月，Trolltech 发布了 GPL 下的 Qt 库。

11 月，IBM 宣布将在 2001 年投资 10 亿美元在 Linux。首部基于 Linux 的手机 IMT-2000 在韩国发布。

这一年的某一天，和同学坐在学校四大发明广场上观看同一首歌演出，困意盎然，期间那个粗犷的名歌星的一句话却惊醒了我：“希望你们交通大学为中国的交通事业做出更大的贡献”，大意如此，我顿时无语，他的语言竟然和他的外表一样粗犷。

这一年的暑假，我第一次来到江南，在西湖断桥对面的饭馆里，透过落地窗恰恰看到湖里荷花的位置，要了份西湖醋鱼和一瓶啤酒，坐到下午四点钟，然后顺着苏堤白堤静静的走

下去，直到绕湖一周再次回到断桥，已是晚上八点，坐在湖边的长凳上，一夜无语。

和平、爱情和 Linux

2001 年

1 月，期待已久的 Linux 2.4 发布。

3 月，Linux2.5 内核高级会议在加州圣何塞举行，它或许是历史上 Linux 内核 hacker 最完整的一次聚会。

4 月，IBM 在几个城市鼓吹“和平、爱情和 Linux”（Peace, Love and Linux）时遇到了麻烦。

6 月，Sharp 宣布基于 Lineo 嵌入式系统的 Linux PDA 即将上市。

这一年底，找工作的季节，我深刻认识了 IT 泡沫和 9.11，找所谓的好工作无门和出国无门，我无奈选择考研。

2002 年

Linus Torvalds 将 Linux 2.4 交由巴西 18 岁的内核开发人员 Marcelo Tosatti 维护，自己则带领 Linux 2.5 的开发工作。

这一年，我从一个交大到了另一个交大，这个转变似乎很平淡，并不深刻。

缅怀已逝的十八年（2003～2006）

Ubuntu 4.10

2003 年

1 月，NEC 宣布将在其手机中使用 Linux，代表着 Linux 成功进军手机领域。

6 月，IDC 分析师称，2003 年 Linux 服务器在西欧的销售量将达到 18.2 万台，到 2007 年，销售量将增至这个数字的三倍，销售收入将翻一番，达到 19 亿美元。

8 月，韩国国家航空公司和 IBM 联合发布声明，表示韩国航空公司将把该公司的核心业务移植到 IBM 的 eServer 服务器当中完成，其中操作系统则采用 Linux。

9 月，三星在推出了首款基于 Linux 系统平台的 CDMA 智能手机 SCH-i519。

11 月，Linux 2.6 发布，它被认为是第一款真正意义上的企业级内核，这是 Linux 内核从 2001 年以来第一次的大改动。

这一年，我第一次在电视直播里看着自己喜欢的米兰夺得了冠军杯。

2004 年

1 月，X.Org 基金会成立。

2 月，Linux 标准 2.0 出台，规范了所有能被称为 Linux 操作系统所应该有的特性。

5 月，基于 Linux 的路由系统出现。

10 月 20 日，Ubuntu 首个版本发布，在五年后的今天 Ubuntu 已经是 Linux 桌面发行版的一个成功典范。

11 月，Firefox 1.0 发布，它成为大众关注的焦点，IE 降低了 1 个点的市场份额——像这种事已经多年没有发生过了。Firefox 已经成为了微软 IE 的强有力的对手。

又到了找工作的季节，宣讲会、笔试、面试，我就要离开学校了么？

2005 年

10 月，Firefox 的下载量突破了 1 亿大关，这表明，只要产品好，开放源代码软件也能够获得普通用户的青睐。

11 月，Sun 开放了除 Java 之外的几乎所有软件，这使得它在一夜间成为了最大的开放源码软件厂商之一。

12 月，Red Hat 公布了第三季度业报，销售收入增长了 43.6%，利润增长了 114%。

这一年夏天，遭遇了到目前为止最为严重的一次失窃，除了 IQ 卡，所有的卡都随着钱夹子消失了，到工行补办牡丹卡时，那慵懒的上海女人说，必须要上海土生土长的本地人来担保，仅仅拥有上海户口的人是不行的。

Richard Stallman 的征婚启事

2006 年

6 月，自由软件之父 Richard Stallman 在自己的网站 http://www.stallman.org/ 上发布了一则“征婚启事”。

I'm a single atheist white man, 52, reputedly intelligent, with unusual interests in politics, science, music and dance.

I'd like to meet a woman with varied interests, curious about the world, comfortable expressing her likes and dislikes (I hate struggling to guess), delighting in her ability to fascinate a man and in being loved tenderly, who values joy, truth, beauty and justice more than "success"--so we can share bouts of intense, passionately kind awareness of each other, alternating with tolerant warmth while we're absorbed in other aspects of life.

My 22-year-old child, the Free Software Movement, occupies most of my life, leaving no room for more children, but I still have room to love a sweetheart. I spend a lot of my time traveling to give speeches, often to Europe, Asia and Latin America; it would be nice if you were free to travel with me some of the time.

If you are interested, write to rms at stallman dot org and we'll see where it leads.

我，单身，无神论者，白人，52 岁，据说比较聪明，对于政治、科学、音乐和舞蹈有着不同寻常的兴趣。

我想寻找这样一位女士：爱好广泛，对世界充满好奇心，能够清晰表达她的爱憎（我痛恨动脑筋猜测），乐于使男人着迷，渴望被温柔地爱，对于快乐、真 理、美和正义的评价高于“成功”。这样的话，我们就能不断对另一方产生热烈而又美好的了解，当我们被生活中其他东西吸引的时候，彼此就能感到宽容的温暖。

我有一个 22 岁的孩子——自由软件运动——他占据了我大部分的生活，没有精力再抚养更多的孩子了，但是我仍然会投入的爱我的爱人。我有大量时间花在巡回演讲上，经常要去欧洲、亚洲和拉丁美洲。如果你有空在某些时间陪我一起旅行，那就最好了。

如果你有兴趣的话，请写信到 rms@stallman.org，让我们看看会有什么结果。

7 月，Ubuntu 被授予 PC World 2006 World Class Award，证明了 Ubuntu 成为 2006 年世界最好的 100 个产品之一。Ubuntu 越来越显示出他的不凡实力，虽说他是免费的，但是后台却是商业公司 Canonical，加上太空人老板的聪明才智，逐渐的开始商业合作，比如和 Sun 合作，对有需要的客户提供 Linux 支持服务。

8 月，Linux 业界另外一位狂人，Linuspire 公司总裁 Kevin Carmony 宣布推出免费版本的 Freespire 1.0，该版本中附带有二进制的商业硬件驱动程序，在 Linux 社区中引起轩然大波。27 日，网站 http://linux.inet.hr/poll_filesystem.html 上推出“Your favorite file system?”（你最喜欢的文件系统？）投票活动。

9 月，16 日是“国际软件自由日”（SFD，Software Freedom Day 2006）。

10 月，Oracle Unbreakable Linux 发布，Oralce 成为第一个推出自有 Linux 服务的非操作系统软件厂商。17 日，FSG（自由标准组，一个非赢利的致力于开发和促进自由开放软件的标准的组织）宣布与 O'Reilly Media 合作，共同为 Linux 应用程序开发人员提供类似 MSDN 的服务，该服务将作为 LSB (Linux Standard Base) Developer Network 的一个组成部分。

11 月，微软和 Novell 达成一揽子协议，号称要改善 Linux 和微软操作系统的兼容问题。如图 1.1 所示，看着昔日的对手用“＋”连起来是否会觉得古怪？

图 1.1 微软 + Novell

这一年，三次去青岛，回来时遭遇三次严重的飞机晚点，让我疑惑这个世界怎么了？

缅怀已逝的十八年（2007～2009）

来自微软的指控

2007 年

1 月，虚拟人生游戏（Second Life）客户端开源。两大 Linux 领导社团 OSDL 和 Free Standard Group 宣布合并为新的 Linux Foundation（Linux 基金会），此举将促进社区的资源整合，也使 Linux 在企业市场能够更加高效地参与竞争。

2 月，Bill Xu 发起了一个“致招商银行的公开信”的行动，希望用这种方式促使招商银行改变在公众服务中使用专属软件的作法，取消客户端上的 ActiveX 技术，而转用其他公开的、开放的、不限制用户平台的技术。据说，浦发银行的网络银行能很好的支持 Firefox。

3 月，Novell 推出模仿苹果的“Mac vs PC”广告，它在广告中插入了第三者：Linux——一位迷人的年轻女子。Novell 用此来宣传预装 Novell Linux 的 PC，一共发布了三个视频，你可以在 www.youtube.com 上看到它们。15 日，Novell 公开表示，同意从总费用上说 Linux 比 Windows 要昂贵，这使它在开源社区的名誉进一步恶化。

4 月，Dell 推出预装 Ubuntu 操作系统笔记本。

5 月，微软声称 Linux 内核侵犯了微软的 42 项专利，而用户界面和其它设计方面也有 65 项侵权，OpenOffice.org 也被指控侵犯 45 项专利，还有 83 项是针对其它免费开源软件。同一个月，微软加拿大网站推出了一个“Get the Facts”（了解真相）页面，如图 1.2 所示，赤裸裸地对 Linux 进行了攻击，有趣的是页面上方放置了一张《The Highly Reliable Times》报纸截图，标题模仿《纽约时报》风格。“报纸”中写道：“我们采用 Linux 平台以后每周至少遭遇一次系统崩溃问题。而迁移到微软 Windows Server 2003 后真正消灭了系统崩溃问题，另外我们还能获得厂商支持。”

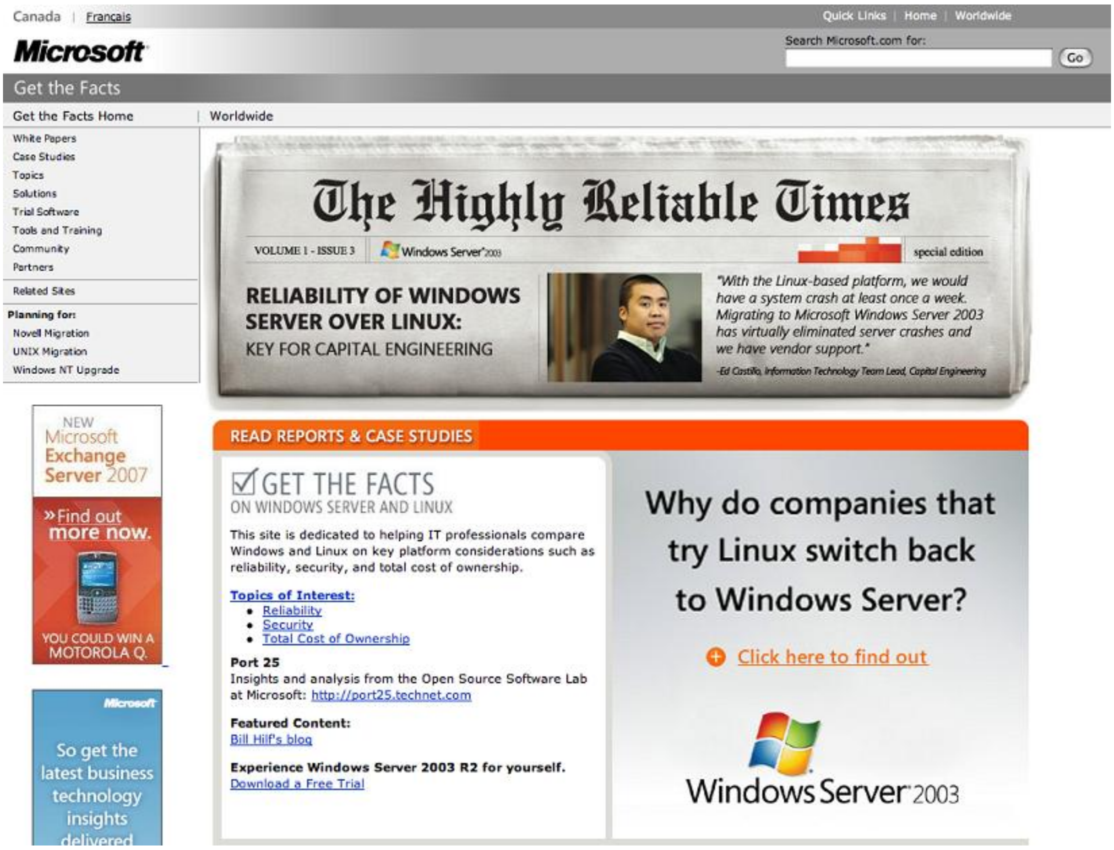


图 1.2 Get the Facts 页面

还是 5 月，Firefox 在 Linux 中显示的表单控件，特别是单选框，比较丑陋问题被修正，如图 1.3 所示。

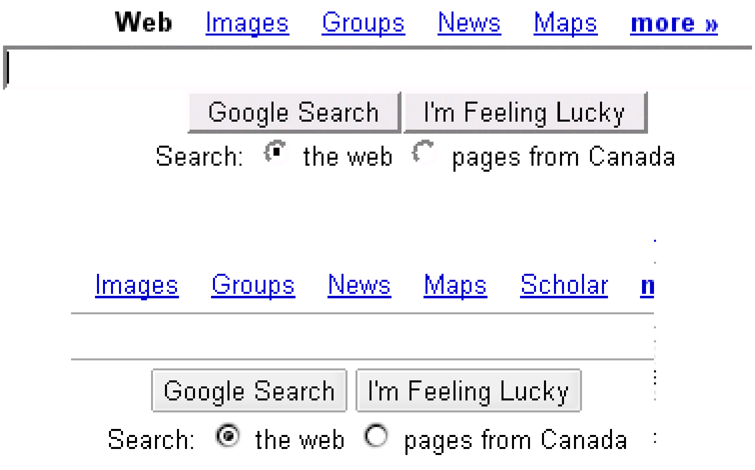


图 1.3 Firefox 表单控件修正前后比较

6 月，5 日微软和 Linux 发行商 Xandros 宣布，双方达成了一个技术和法律上的合作。Red Hat、Ubuntu 与 Mandriva 拒绝与微软进行专利交易。28 日，Google 桌面搜索 Linux 版正式发布，截图如图 1.4 所示。29 日，第三版 GNU 通用公共许可证 GPLv3 发布。

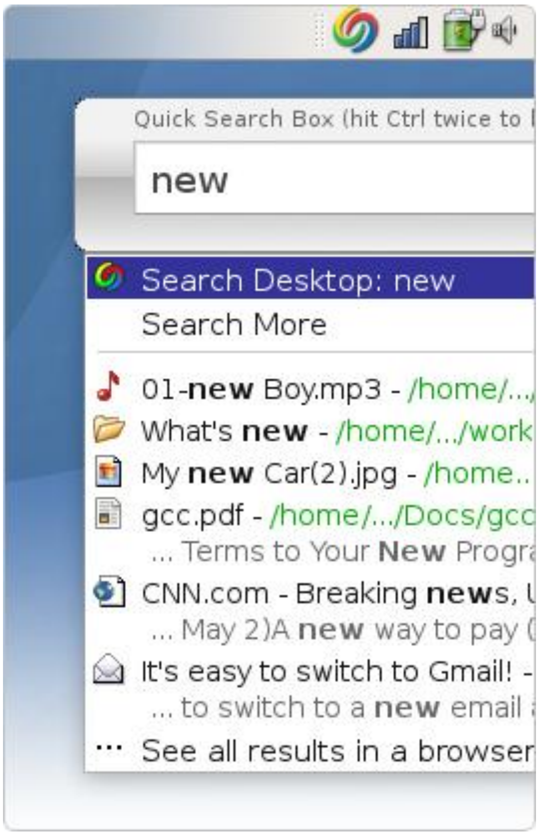


图 1.4 Google 桌面搜索 Linux 版

7 月，Fcitx 小企鹅输入法开源项目终止。做为 Linux 平台上最受欢迎的两大中文输入法之一——Fcitx 小企鹅输入法，在其官方主页上宣布项目终止开发。声明中提到，有“编程高手”质疑其代码风格是项目终止的导火索。

8 月，SCO 在控告 Linux 侵犯专利权的官司中败诉，从而申请破产保护。SCO 面市时以 Linux 销售商 Caldera Systems 的面目现身，然后从 Santa Cruz Operation 收购了 Unix 业务，之后重名为 SCO 集团。然后他们放弃了 Linux 业务，并开始起诉 IBM、Novell 及其他公司。他们认为 IBM 破坏了他们签署的 Unix 协议，将 SCO 特有的 Unix 技术在开源的 Linux 社区发布出去。Jim Zemlin 对此评论说：“如果它们把事业基础建立在协助 Linux，而不是攻击 Linux，那么它们大可享有像 RedHat 这些公司一样的成功，而不是沦落到申请破产保护的下场。”

10 月，Acacia Research 通过其子公司 IP Innovation 向 RedHat 和 Novell 提出控告，RedHat Linux 操作系统及 Novell 旗下的 SUSE Linux Enterprise Desktop 与 SUSE Linux Enterprise Server 侵犯了他所拥有的专利。随着围绕开源的纠纷不断，2007 年对于律师来说注定是“丰收”的一年。

11 月，Google 推出基于 Linux 的开源移动平台 Android。Phoronix 网站发布了 ATI 显卡在 Linux 和 Vista 下的游戏性能对比测试，结果令人鼓舞，在 Linux 下的游戏性能首次超越了 Windows！

这一年，我开始在 blog.csdn.net/fudan_abc 上写《Linux 那些事儿》。

首款 Android 手机

2008 年

1 月，Nokia 宣布收购了著名开源跨平台开发工具 Qt 的开发商 Trolltech。

2 月，Google 资助 Linux 版 Photoshop 的研究。

4 月，Sun 移除了 Java 最后的限制，将其彻底开源。

7 月，腾讯公司在这个月的最后一天发布了 QQ for Linux 1.0 Preview 版，这是第一次官方的版本。

9 月，Google 联合 T-Mobile、HTC，正式发布了首款 Android 平台的手机 G1。Google 开源浏览器 Chrome 发布，发布仅仅几个小时，它的总体占有率就达到了 2%。

10 月，OpenOffice3.0 发布，这对 Linux 的普及和实用化影响巨大。月底 Ubuntu 8.10 发布，Fedora 10 发布。

12 月，各类发行版的 Linux 操作系统占据了大约三成的上网本市场份额。

对于我来说，这一年的基调是出差，大半年之后回到上海，很多地方很多事情都变得陌生起来，才发觉忘却其实也是一件很容易的事情。

Linux 信用卡

2009 年

1 月，Linux 兼容内核正式使用项目 Unix 名称 Longene，中文别名“龙井”。兼容内核是一个自由、开源的操作系统项目，目的是要把 Linux 的内核扩充成一个既支持 Linux 应用、也支持 Windows 应用，既支持 Linux 设备驱动、也支持 Windows 设备驱动的兼容内核，使用户可以直接在 Linux 操作系统上高效运行 Windows 应用。

2 月，微软起诉 GPS 设备制造商 Tomtom 侵犯其 8 项专利权，Tomtom 的 GPS 设备采用的是 Linux 系统，尽管微软声称 Linux 侵犯其专利期已有多年，但该案被视为微软状告 Linux 侵权的第一案。

3 月，Adobe Reader 9.1 for Linux 发布。UltraEdit 正被移植到 Linux，名为 UEX，意即 UltraEdit for Linux。

4 月，IDC 最新发表的题为《Linux 在新经济中的机会》的报告称，用户 2009 年的 Linux 开支预计将比 2008 年增长 21%，超过整个软件市场的增长速度。整个软件市场 2009 年的增长率是 2%。

5 月，Nokia 宣布开放 Qt 源代码仓库，以便让社区的开发者能够进一步参与 Qt 的开发。

6 月，法国的 ENAC 开发组为 Linux 内核开发了类似 iPhone 的 Multi-touch（多点触摸）技术支持。

7 月，Linux 基金会与 CardPartner 和 UMBrella 银行共同推出了 Visa 白金信用卡，正面印有 Linux 吉祥物，如图 1.5 所示。每办理一张这样的信用卡就可以为 Linux 基金会带来 50 美元的赞助，使用该卡每消费一次 Linux 基金会就能从中获得 1%的金额。

图 1.5 Linux 信用卡

8 月，微软在提交给美国证券交易委员会的年度文件中，将 Unbutu 列入竞争对手。

9 月，目前为止最新的内核版本 2.6.31 发布，Linux 成为首个正式支持 USB3.0 的操作系统。

10 月，Ubuntu 9.10 发布。

11 月，Vim 的作者 Bram Moolenaar 推出了新的编程语言 Zimbu，一种不拐弯抹角直截了当的实验性编程语言。Moolenaar 表示 Zimbu 集现有语言的优点于一身， 同时避开它们的不足。Zimbu 代码清晰易读，使用范围广泛——既能写 OS kernel，又能写脚本，还能写大的 GUI 程序，可以编译和运行在几乎所有系统上。

Kernel 地图：Kconfig 与 Makefile

Makefile 不是 Make Love

从前在学校，混了四年，没有学到任何东西，每天就是逃课，上网，玩游戏，睡觉。毕业的时候，人家跟我说 Makefile 我完全不知，但是一说 Make Love 我就来劲了，现在想来依然觉得丢人。

毫不夸张地说，Kconfig 和 Makefile 是我们浏览内核代码时最为依仗的两个文件。基本上，Linux 内核中每一个目录下边都会有一个 Kconfig 文件和一个 Makefile 文件。 对于一个希望能够在 Linux 内核的汪洋代码里看到一丝曙光的人来说，将它们放在怎么重要的地位都不过分。

我们去香港，通过海关的时候，总会有免费的地图和各种指南拿，有了它们在手里我们才不至于无头苍蝇般迷惘的行走在陌生的街道上。即使在内地出去旅游的时候一般来说也总是会首先找份地图，当然了，这时就是要去买了，拿是拿不到的，不同的地方有不同的特色，只不过有的特色是服务，有的特色是索取。

Kconfig 和 Makefile 就是 Linux Kernel 迷宫里的地图。地图引导我们去认识一个城市，而 Kconfig 和 Makefile 则可以让我们了解一个 Kernel 目录下面的结构。我们每次浏览 kernel 寻找属于自己的那一段代码时，都应该首先看看目录下的这两个文件。

利用 Kconfig 和 Makefile 寻找目标代码

就像利用地图寻找目的地一样，我们需要利用 Kconfig 和 Makefile 来寻找所要研究的目标代码。

比如我们打算研究 U 盘驱动的实现，因为 U 盘是一种 storage 设备，所以我们应该先进入到 drivers/usb/storage/目录。但是该目录下的文件很多，那么究竟哪些文件才是我们需要关注的？这时就有必要先去阅读 Kconfig 和 Makefile 文件。

对于 Kconfig 文件，我们可以看到下面的选项。

```
config USB_STORAGE_DATAFAB
     bool "Datafab Compact Flash Reader support (EXPERIMENTAL)"
     depends on USB_STORAGE && EXPERIMENTAL
     help
       Support for certain Datafab CompactFlash readers.
       Datafab has a web page at <http://www.datafabusa.com/>.
```

显然，这个选项和我们的目的没有关系。首先它专门针对 Datafab 公司的产品，其次虽然 CompactFlash reader 是一种 flash 设备，但显然不是 U 盘。因为 drivers/usb/storage 目录下的代码是针对 usb mass storage 这一类设备，而不是针对某一种特定的设备。U 盘只是 usb mass storage 设备中的一种。再比如：

```
config USB_STORAGE_SDDR55
     bool "SanDisk SDDR-55 SmartMedia support (EXPERIMENTAL)"
     depends on USB_STORAGE && EXPERIMENTAL
     help
       Say Y here to include additional code to support the Sandisk SDDR-55
       SmartMedia reader in the USB Mass Storage driver.
```

很显然这个选项是有关 SanDisk 产品的，并且针对的是 SM 卡，同样不是 U 盘，所以我们也不需要去关注。

事实上，很容易确定，只有选项 CONFIG_USB_STORAGE 才是我们真正需要关注的。

```
config USB_STORAGE
 tristate "USB Mass Storage support"
     depends on USB && SCSI
 ---help---
   Say Y here if you want to connect USB mass storage devices to your
   computer's USB port. This is the driver you need for USB
   floppy drives, USB hard disks, USB tape drives, USB CD-ROMs,
   USB flash devices, and memory sticks, along with
   similar devices. This driver may also be used for some cameras
   and card readers.

   This option depends on 'SCSI' support being enabled, but you
```

```
     probably also need 'SCSI device support: SCSI disk support'
  (BLK_DEV_SD) for most USB storage devices.

  To compile this driver as a module, choose M here: the
  module will be called usb-storage.
```

接下来阅读 Makefile 文件。

```
 0 #
 1 # Makefile for the USB Mass Storage device drivers.
 2 #
 3 # 15 Aug 2000, Christoph Hellwig <hch@infradead.org>
 4 # Rewritten to use lists instead of if-statements.
 5 #
 6
 7 EXTRA_CFLAGS := -Idrivers/scsi
 8
 9 obj-$(CONFIG_USB_STORAGE)    += usb-storage.o
10
11 usb-storage-obj-$(CONFIG_USB_STORAGE_DEBUG) += debug.o
12 usb-storage-obj-$(CONFIG_USB_STORAGE_USBAT) += shuttle_usbat.o
13 usb-storage-obj-$(CONFIG_USB_STORAGE_SDDR09)    += sddr09.o
14 usb-storage-obj-$(CONFIG_USB_STORAGE_SDDR55)    += sddr55.o
15 usb-storage-obj-$(CONFIG_USB_STORAGE_FREECOM)   += freecom.o
16 usb-storage-obj-$(CONFIG_USB_STORAGE_DPCM)  += dpcm.o
17 usb-storage-obj-$(CONFIG_USB_STORAGE_ISD200)    += isd200.o
18 usb-storage-obj-$(CONFIG_USB_STORAGE_DATAFAB)   += datafab.o
19 usb-storage-obj-$(CONFIG_USB_STORAGE_JUMPSHOT)  += jumpshot.o
20 usb-storage-obj-$(CONFIG_USB_STORAGE_ALAUDA)    += alauda.o
21 usb-storage-obj-$(CONFIG_USB_STORAGE_ONETOUCH)  += onetouch.o
22 usb-storage-obj-$(CONFIG_USB_STORAGE_KARMA) += karma.o
23
24 usb-storage-objs :=  scsiglue.o protocol.o transport.o usb.o \
25          initializers.o $(usb-storage-obj-y)
26
27 ifneq ($(CONFIG_USB_LIBUSUAL),)
28  obj-$(CONFIG_USB)   += libusual.o
29 endif
```

前面通过 Kconfig 文件的分析，我们确定了只需要去关注 CONFIG_USB_STORAGE 选项。在 Makefile 文件里查找 CONFIG_USB_STORAGE，从第 9 行得知，该选项对应的模块为 usb-storage。

因为 Kconfig 文件里的其他选项我们都不需要关注,所以 Makefile 的 11~22 行可以忽略。第 24 行意味着我们只需要关注 scsiglue.c、protocol.c、transport.c、usb.c、initializers.c 以及它们同名的.h 头文件。

Kconfig 和 Makefile 很好的帮助我们定位到了所要关注的目标，就像我们到一个陌生的地方要随身携带地图，当我们学习 Linux 内核时，也要谨记寻求 Kconfig 和 Makefile 的帮助。

分析内核源码如何入手？（上）

透过现象看本质，兽兽门无非就是一些人体艺术展示。同样往本质里看过去，学习内核，就是学习内核的源代码，任何内核有关的书籍都是基于内核，而又不高于内核的。

既然要学习内核源码，就要经常对内核代码进行分析，而内核代码千千万，还前仆后继的不断往里加，这就让大部分人都有种雾里看花花不见的无助感。不过不要怕，孔老夫子早就留给我们了应对之策：敏于事而慎于言，就有道而正焉，可谓好学也已。这就是说，做事要踏实才是好学生好同志，要遵循严谨的态度，去理解每一段代码的实现，多问多想多记。如果抱着走马观花，得过且过的态度，结果极有可能就是一边看一边丢，没有多大的收获。

假设全国房价上涨 1.5%，假设 80 后局长是农民子弟，……，既然我们的人生充满了假设，那么我在这里假设你现在就迫不及待的希望研究内核中 USB 子系统的实现，应该没有意见吧？那好，下面就以 USB 子系统的实现分析为标本看看分析内核源码应该如何入手。

分析 README

内核中 USB 子系统的代码位于目录 drivers/usb，这个结论并不需要假设。于是我们进入到该目录，执行命令 ls，结果显示如下：

```
atm  class  core  gadget  host  image  misc  mon  serial  storage Kconfig
Makefile  README usb-skeleton.c
```

目录 drivers/usb 共包含有 10 个子目录和 4 个文件，usb-skeleton.c 是一个简单的 USB driver 的框架，感兴趣的可以去看看，目前来说，它还吸引不了我们的眼球。那么首先应该关注什么？如果迎面走来一个 ppmm，你会首先看脸、脚还是其它？当然答案依据每个人的癖好会有所不同。不过这里的问题应该只有一个答案，那就是 Kconfig、Makefile、README。

README 里有关于这个目录下内容的一般性描述，它不是关键，只是帮助你了解。再说了，面对“read 我吧 read 我吧”这么热情奔放的呼唤，善良的我们是不可能无动于衷的，所以先来看看里面都有些什么内容。

```
Here is a list of what each subdirectory here is, and what is contained in
them.

core/         - This is for the core USB host code, including the
              usbfs files and the hub class driver ("khubd").

host/         - This is for USB host controller drivers.  This
              includes UHCI, OHCI, EHCI, and others that might
              be used with more specialized "embedded" systems.

gadget/       - This is for USB peripheral controller drivers and
```

```
            the various gadget drivers which talk to them.


Individual USB driver directories.  A new driver should be added to the
first subdirectory in the list below that it fits into.

image/        - This is for still image drivers, like scanners or
            digital cameras.
input/        - This is for any driver that uses the input subsystem,
            like keyboard, mice, touchscreens, tablets, etc.
media/        - This is for multimedia drivers, like video cameras,
            radios, and any other drivers that talk to the v4l
            subsystem.
net/      - This is for network drivers.
serial/       - This is for USB to serial drivers.
storage/ - This is for USB mass-storage drivers.
class/        - This is for all USB device drivers that do not fit
            into any of the above categories, and work for a range
            of USB Class specified devices.
misc/         - This is for all USB device drivers that do not fit
            into any of the above categories.
```

这个 README 文件描述了前边使用 ls 命令列出的那 10 个文件夹的用途。那么什么是 USB Core？Linux 内核开发者们，专门写了一些代码，负责实现一些核心的功能，为别的设备驱动程序提供服务，比如申请内存，比如实现一些所有的设备都会需要的公共的函数，并美其名曰 USB Core。

时代总在发展，当年胖杨贵妃照样迷死唐明皇，而如今人们欣赏的则是林志玲这样的魔鬼身材。同样，早期的 Linux 内核，其结构并不是如今天这般有层次感，远不像今天这般错落有致，那时候 drivers/usb/这个目录下边放了很多很多文件，USB Core 与其他各种设备的驱动程序的代码都堆砌在这里，后来，怎奈世间万千的变幻，总爱把有情的人分两端。于是在 drivers/usb/目录下面出来了一个 core 目录，就专门放一些核心的代码，比如初始化整个 USB 系统，初始化 Root Hub，初始化主机控制器的代码，再后来甚至把主机控制器相关的代码也单独建了一个目录，叫 host 目录，这是因为 USB 主机控制器随着时代的发展，也开始有了好几种，不再像刚开始那样只有一种，所以呢，设计者们把一些主机控制器公共的代码仍然留在 core 目录下，而一些各主机控制器单独的代码则移到 host 目录下面让负责各种主机控制器的人去维护。

那么 USB gadget 那？gadget 白了说就是配件的意思，主要就是一些内部运行 Linux 的嵌入式设备，比如 PDA，设备本身有 USB 设备控制器（USB Device Controller），可以将 PC，也就是我们的主机作为 master 端，将这样的设备作为 slave 端和主机通过 USB 进行通信。从主机的观点来看，主机系统的 USB 驱动程序控制插入其中的 USB 设备，而 USB gadget 的驱动程序控制外围设备如何作为一个 USB 设备和主机通信。比如，我们的嵌入式板子上支持 SD 卡，如果我们希望在将板子通过 USB 连接到 PC 之后，这个 SD 卡被模拟成 U 盘，那么就要通过 USB gadget 架构的驱动。

剩下的几个目录分门别类的放了各种 USB 设备的驱动，比如 U 盘的驱动在 storage 目录下，触摸屏和 USB 键盘鼠标的驱动在 input 目录下，等等。

我们响应了 README 的热情呼唤，它便给予了我们想要的，通过它我们了解了 USB 目录里的那些文件夹都有着什么样的角色。到现在为止，就只剩下内核的地图——Kconfig 与 Makefile 两个文件了。有地图在手，对于在内核中游荡的我们来说，是件很愉悦的事情，不过，因为我们的目的是研究内核对 USB 子系统的实现，而不是特定设备或 host controller 的驱动，所以这里的定位很明显，USB Core 就是我们需要关注的对象，那么接下来就是要对 core 目录中的内容进行定位了。

分析 Kconfig 和 Makefile

进入到 drivers/usb/core 目录，执行命令 ls，结果显示如下：

```
Kconfig  Makefile  buffer.c  config.c  devices.c  devio.c  driver.c
endpoint.c  file.c  generic.c  hcd-pci.c  hcd.c  hcd.h  hub.c  hub.h
inode.c  message.c  notify.c  otg_whitelist.h  quirks.c  sysfs.c  urb.c
usb.c  usb.h
```

然后执行 wc 命令，如下所示。

```
# wc -l ./*
   148 buffer.c
   607 config.c
   706 devices.c
  1677 devio.c
  1569 driver.c
   357 endpoint.c
   248 file.c
   238 generic.c
  1759 hcd.c
   458 hcd.h
   433 hcd-pci.c
  3046 hub.c
   195 hub.h
   758 inode.c
   144 Kconfig
    21 Makefile
  1732 message.c
    68 notify.c
   112 otg_whitelist.h
   161 quirks.c
   710 sysfs.c
   589 urb.c
   984 usb.c
   160 usb.h
 16880 total
```

drivers/usb/core 目录共包括 24 个文件，16880 行代码。core 不愧是 core，为大家默默的

做这么多事。不过这么多文件里不一定都是我们所需要关注的，先拿咱们的地图来看看接下来该怎么走。先看看 Kconfig 文件，可以看到下面的选项。

```
config USB_DEVICEFS
        bool "USB device filesystem"
        depends on USB
        ---help---
          If you say Y here (and to "/proc file system support" in the "File
                      systems" section, above), you will get a file /proc/bus/usb/devices
          which lists the devices currently connected to your USB bus or
          busses, and for every connected device a file named
          "/proc/bus/usb/xxx/yyy", where xxx is the bus number and yyy the
          device number; the latter files can be used by user space programs
          to talk directly to the device. These files are "virtual", meaning
          they are generated on the fly and not stored on the hard drive.

          You may need to mount the usbfs file system to see the files, use
          mount -t usbfs none /proc/bus/usb

          For the format of the various /proc/bus/usb/ files, please read
          <file:Documentation/usb/proc_usb_info.txt>.

          Usbfs files can't handle Access Control Lists (ACL), which are the
          default way to grant access to USB devices for untrusted users of a
          desktop system. The usbfs functionality is replaced by real
          device-nodes managed by udev. These nodes live in /dev/bus/usb and
          are used by libusb.
```

选项 USB_DEVICEFS 与 usbfs 文件系统有关。usbfs 文件系统挂载在/proc/bus/usb 目录，显示了当前连接的所有 USB 设备及总线的各种信息，每个连接的 USB 设备在其中都会有一个对应的文件进行描述。比如文件/proc/bus/usb/xxx/yyy，xxx 表示总线的序号，yyy 表示设备所在总线的地址。不过不能够依赖它们来稳定地访问设备，因为同一设备两次连接对应的描述文件可能会不同，比如，第一次连接一个设备时，它可能是 002/027，一段时间后再次连接，它可能就已经改变为 002/048。

就好比好不容易你暗恋的 mm 今天见你的时候对你抛了个媚眼，你心花怒放，赶快去买了 100 块彩票庆祝，到第二天再见到她的时候，她对你说你是谁啊，你悲痛欲绝的刮开那 100 块彩票，上面清一色的谢谢你。

因为 usbfs 文件系统并不属于 USB 子系统实现的核心部分，与之相关的代码我们可以不必关注。

```
config USB_SUSPEND
      bool "USB selective suspend/resume and wakeup (EXPERIMENTAL)"
```

```
        depends on USB && PM && EXPERIMENTAL
        help
          If you say Y here, you can use driver calls or the sysfs
          "power/state" file to suspend or resume individual USB
          peripherals.

          Also, USB "remote wakeup" signaling is supported, whereby some
          USB devices (like keyboards and network adapters) can wake up
          their parent hub.  That wakeup cascades up the USB tree, and
          could wake the system from states like suspend-to-RAM.

          If you are unsure about this, say N here.
```

这一项是有关 USB 设备的挂起和恢复。开发 USB 的人都是节电节能的好孩子，所以协议里就规定了，所有的设备都必须支持挂起状态，就是说为了达到节电的目的，当设备在指定的时间内，如果没有发生总线传输，就要进入挂起状态。当它收到一个 non-idle 的信号时，就会被唤醒。节约用电从 USB 做起。不过这个与主题也没太大关系，相关代码也可以不用关注了。

剩下的还有几项，不过似乎与咱们关系也不大，还是去看看 Makefile。

```
usbcore-objs    := usb.o hub.o hcd.o urb.o message.o driver.o \
                        config.o file.o buffer.o sysfs.o endpoint.o \
                        devio.o notify.o generic.o quirks.o

ifeq ($(CONFIG_PCI),y)
        usbcore-objs    += hcd-pci.o
endif

ifeq ($(CONFIG_USB_DEVICEFS),y)
        usbcore-objs    += inode.o devices.o
endif

obj-$(CONFIG_USB)       += usbcore.o

ifeq ($(CONFIG_USB_DEBUG),y)
EXTRA_CFLAGS += -DDEBUG
endif
```

Makefile 可比 Kconfig 简略多了，所以看起来也更亲切点，咱们总是拿的 money 越多越好，看的代码越少越好。这里之所以会出现 CONFIG_PCI，是因为通常 USB 的 Root Hub 包含在一个 PCI 设备中。hcd-pci 和 hcd 顾名而思义就知道是说主机控制器的，它们实现了主机控制器公共部分，按协议里的说法它们就是 HCDI（HCD 的公共接口），host 目录下则实现了各种不同的主机控制器。CONFIG_USB_DEVICEFS 前面的 Kconfig 文件里也见到了，关于 usbfs 的，与咱们的主题无关，inode.c 和 devices.c 两个文件也可以不用管了。

那么我们可以得出结论，为了理解内核对 USB 子系统的实现，我们需要研究 buffer.c、

config.c、driver.c、endpoint.c、file.c、generic.c、hcd.c hcd.h、hub.c、message.c、notify.c、otg_whitelist.h、quirks.c、sysfs.c、urb.c 和 usb.c 文件。这么看来，好像大都需要关注的样子，没有减轻多少压力，不过这里本身就是 USB Core 部分，是要做很多的事为咱们分忧的，所以多点也是可以理解的。

分析内核源码如何入手？（下）

下面的分析，米卢教练说了，内容不重要，重要的是态度。就像韩局长对待日记的态度那样，严谨而细致。

只要你使用这样的态度开始分析内核，那么无论你选择内核的哪个部分作为切入点，比如 USB，比如进程管理，在花费相对不算很多的时间之后，你就会发现你对内核的理解会上升到另外一个高度，一个抱着情景分析，抱着 0.1 内核完全注释，抱着各种各样的内核书籍翻来覆去的看很多遍又忘很多遍都无法达到的高度。请相信我！

让我们在 Linux 社区里发出号召：学习内核源码，从学习韩局长开始！

态度决定一切：从初始化函数开始

任小强们说房价高涨从现在开始，股评家们说牛市从 5000 点开始。他们的开始需要我们的钱袋，我们的开始只需要一台电脑，最好再有一杯茶，伴着几支小曲儿，不盯着钱总是会比较惬意的。生容易，活容易，生活不容易，因为总要盯着钱。

有了地图 Kconfig 和 Makefile，我们可以在庞大复杂的内核代码中定位以及缩小了目标代码的范围。那么现在，为了研究内核对 USB 子系统的实现，我们还需要在目标代码中找到一个突破口，这个突破口就是 USB 子系统的初始化代码。

针对某个子系统或某个驱动，内核使用 subsys_initcall 或 module_init 宏指定初始化函数。在 drivers/usb/core/usb.c 文件中，我们可以发现下面的代码。

```
subsys_initcall(usb_init);
module_exit(usb_exit);
```

我们看到一个 subsys_initcall，它也是一个宏，我们可以把它理解为 module_init，只不过因为这部分代码比较核心，开发者们把它看作一个子系统，而不仅仅是一个模块。这也很好理解，usbcore 这个模块它代表的不是某一个设备，而是所有 USB 设备赖以生存的模块，Linux 中，像这样一个类别的设备驱动被归结为一个子系统。比如 PCI 子系统，比如 SCSI 子系统，基本上，drivers/目录下面第一层的每个目录都算一个子系统，因为它们代表了一类设备。

subsys_initcall(usb_init)的意思就是告诉我们 usb_init 是 USB 子系统真正的初始化函数，而 usb_exit()将是整个 USB 子系统的结束时的清理函数。于是为了研究 USB 子系统在内核中的实现，我们需要从 usb_init 函数开始看起。

```
static int __init usb_init(void)
{
    int retval;
    if (nousb) {
        pr_info("%s: USB support disabled\n", usbcore_name);
```

```
		return 0;
	}

	retval = ksuspend_usb_init();
	if (retval)
		goto out;
	retval = bus_register(&usb_bus_type);
	if (retval)
		goto bus_register_failed;
	retval = usb_host_init();
	if (retval)
		goto host_init_failed;
	retval = usb_major_init();
	if (retval)
		goto major_init_failed;
	retval = usb_register(&usbfs_driver);
	if (retval)
		goto driver_register_failed;
	retval = usb_devio_init();
	if (retval)
		goto usb_devio_init_failed;
	retval = usbfs_init();
	if (retval)
		goto fs_init_failed;
	retval = usb_hub_init();
	if (retval)
		goto hub_init_failed;
	retval = usb_register_device_driver(&usb_generic_driver, THIS_MODULE);
	if (!retval)
		goto out;

	usb_hub_cleanup();
hub_init_failed:
	usbfs_cleanup();
fs_init_failed:
	usb_devio_cleanup();
usb_devio_init_failed:
	usb_deregister(&usbfs_driver);
driver_register_failed:
	usb_major_cleanup();
major_init_failed:
	usb_host_cleanup();
host_init_failed:
```

```
913     bus_unregister(&usb_bus_type);
914 bus_register_failed:
915     ksuspend_usb_cleanup();
916 out:
917     return retval;
918 }
```

（1）__init 标记。

关于 usb_init，第一个问题是，第 865 行的__init 标记具有什么意义？

写过驱动的应该不会陌生，它对内核来说就是一种暗示，表明这个函数仅在初始化期间使用，在模块被装载之后，它占用的资源就会释放掉用作它处。它的暗示你懂，可你的暗示，她却不懂或者懂装不懂，多么让人感伤。它在自己短暂的一生中一直从事繁重的工作，吃的是草吐出的是牛奶，留下的是整个 USB 子系统的繁荣。

受这种精神所感染，我觉得有必要为它说的更多些。__init 的定义在 include/linux/init.h 文件里

```
43 #define __init          __attribute__ ((__section__ (".init.text")))
```

好像这里引出了更多的疑问，__attribute__是什么？Linux 内核代码使用了大量的 GNU C 扩展，以至于 GNU C 成为能够编译内核的唯一编译器，GNU C 的这些扩展对代码优化、目标代码布局、安全检查等方面也提供了很强的支持。而__attribute__就是这些扩展中的一个，它主要被用来声明一些特殊的属性，这些属性主要被用来指示编译器进行特定方面的优化和更仔细的代码检查。GNU C 支持十几个属性，section 是其中的一个，我们查看 GCC 的手册可以看到下面的描述

```
'section ("section-name")'
Normally, the compiler places the code it generates in the `text'
    section.  Sometimes, however, you need additional sections, or you
need certain particular functions to appear in special sections.
The `section' attribute specifies that a function lives in a
particular section.  For example, the declaration:

     extern void foobar (void) __attribute__ ((section ("bar")));

puts the function 'foobar' in the 'bar' section.

Some file formats do not support arbitrary sections so the
'section' attribute is not available on all platforms.  If you
need to map the entire contents of a module to a particular
section, consider using the facilities of the linker instead.
```

通常编译器将函数放在.text 节，变量放在.data 或.bss 节，使用 section 属性，可以让编译器将函数或变量放在指定的节中。那么前面对__init 的定义便表示将它修饰的代码放在.init.text 节。连接器可以把相同节的代码或数据安排在一起，比如__init 修饰的所有代码都会被放在.init.text 节里，初始化结束后就可以释放这部分内存。

问题可以到此为止，也可以更深入，即内核又是如何调用到这些__init 修饰的初始化函数？要回答这个问题，还需要回顾一下 subsys_initcall 宏，它也在 include/linux/init.h 里定义

```
125 #define subsys_initcall(fn)           __define_initcall("4",fn,4)
```

这里又出现了一个宏__define_initcall，它用于将指定的函数指针 fn 放到 initcall.init 节里而对于具体的 subsys_initcall 宏，则是把 fn 放到.initcall.init 的子节.initcall4.init 里。要弄清楚.initcall.init、.init.text 和.initcall4.init 这样的东东，我们还需要了解一点内核可执行文件相关的概念。

内核可执行文件由许多链接在一起的对象文件组成。对象文件有许多节，如文本、数据、init 数据、bass 等等。这些对象文件都是由一个称为链接器脚本的文件链接并装入的。这个链接器脚本的功能是将输入对象文件的各节映射到输出文件中；换句话说，它将所有输入对象文件都链接到单一的可执行文件中，将该可执行文件的各节装入到指定地址处。vmlinux.lds 是存在于 arch/<target>/ 目录中的内核链接器脚本，它负责链接内核的各个节并将它们装入内存中特定偏移量处。

我可以负责任的告诉你，要看懂 vmlinux.lds 这个文件是需要一番功夫的，不过大家都是聪明人，聪明人做聪明事，所以你需要做的只是搜索 initcall.init，然后便会看到似曾相识的内容

```
__inicall_start = .;
.initcall.init : AT(ADDR(.initcall.init) - 0xC0000000) {
*(.initcall1.init)
*(.initcall2.init)
*(.initcall3.init)
*(.initcall4.init)
*(.initcall5.init)
*(.initcall6.init)
*(.initcall7.init)
}
__initcall_end = .;
```

这里的__initcall_start 指向.initcall.init 节的开始，__initcall_end 指向它的结尾。而.initcall.init 节又被分为了 7 个子节，分别是

```
.initcall1.init

.initcall2.init

.initcall3.init
.initcall4.init
.initcall5.init
.initcall6.init
.initcall7.init
```

我们的 subsys_initcall 宏便是将指定的函数指针放在了.initcall4.init 子节。其它的比如 core_initcall 将函数指针放在.initcall1.init 子节，device_initcall 将函数指针放在了.initcall6.init 子节等等，都可以从 include/linux/init.h 文件找到它们的定义。各个字节的顺序是确定的，即先调用.initcall1.init 中的函数指针再调用.initcall2.init 中的函数指针，等等。__init 修饰的

初始化函数在内核初始化过程中调用的顺序和.initcall.init 节里函数指针的顺序有关，不同的初始化函数被放在不同的子节中，因此也就决定了它们的调用顺序。

至于实际执行函数调用的地方，就在/init/main.c 文件里，内核的初始化么，不在那里还能在哪里，里面的 do_initcalls 函数会直接用到这里的__initcall_start、__initcall_end 来进行判断。

（2）模块参数。

关于 usb_init 函数，第二个问题是，第 868 行的 nousb 表示什么？

知道 C 语言的人都会知道 nousb 是一个标志，只是不同的标志有不一样的精彩，这里的 nousb 是用来让我们在启动内核的时候通过内核参数去掉 USB 子系统的，Linux 社会是一个很人性化的世界，它不会去逼迫我们接受 USB，一切都只关乎我们自己的需要。不过我想我们一般来说是不会去指定 nousb 的吧。如果你真的指定了 nousb，那它就只会幽怨的说一句“USB support disabled”，然后退出 usb_init。

nousb 在 drivers/usb/core/usb.c 文件中定义为：

```
static int nousb;   /* Disable USB when built into kernel image */
module_param_named(autosuspend, usb_autosuspend_delay, int, 0644);
MODULE_PARM_DESC(autosuspend, "default autosuspend delay");
```

从中可知 nousb 是个模块参数。关于模块参数，我们都知道可以在加载模块的时候可以指定，但是如何在内核启动的时候指定？

打开系统的 grub 文件，然后找到 kernel 行，比如：

```
kernel    /boot/vmlinuz-2.6.18-kdb   root=/dev/sda1   ro   splash=silent
vga=0x314
```

其中的 root，splash，vga 等都表示内核参数。当某一模块被编译进内核的时候，它的模块参数便需要在 kernel 行来指定，格式为“模块名.参数=值”，比如：

```
modprobe usbcore autosuspend=2
```

对应到 kernel 行，即为：

```
usbcore.autosuspend=2
```

通过命令“modinfo -p ${modulename}”可以得知一个模块有哪些参数可以使用。同时，对于已经加载到内核里的模块，它们的模块参数会列举在/sys/module/${modulename}/parameters/ 目录下面，可以使用“echo -n ${value} > /sys/module/${modulename}/parameters/${parm}”这样的命令去修改。

（3）可变参数宏。

关于 usb_init 函数，第三个问题是，pr_info 如何实现与使用？

pr_info 只是一个打印信息的可辨参数宏，printk 的变体，在 include/linux/kernel.h 里定义：

```
#define pr_info(fmt,arg...) \
        printk(KERN_INFO fmt,##arg)
```

99 年的 ISO C 标准里规定了可变参数宏，和函数语法类似，比如

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
```

里面的“…”就表示可变参数，调用时，它们就会替代宏体里的__VA_ARGS__。GCC 总是会显得特立独行一些，它支持更复杂的形式，可以给可变参数取个名字，比如

```
#define debug(format, args...) fprintf (stderr, format, args)
```

有了名字总是会容易交流一些。是不是与 pr_info 比较接近了？除了‘##’，它主要是针对空参数的情况。既然说是可变参数，那传递空参数也总是可以的，空即是多，多即是空，股市里的哲理这里同样也是适合的。如果没有‘##’，传递空参数的时候，比如

```
debug ("A message");
```

展开后，里面的字符串后面会多个多余的逗号。这个逗号你应该不会喜欢，而‘##’则会使预处理器去掉这个多余的逗号。

关于 usb_init 函数，上面的三个问题之外，余下的代码分别完成 usb 各部分的初始化，接下来就需要围绕它们分别进行深入分析。因为这里只是演示如何入手分析，展示的只是一种态度，所以具体的深入分析就免了吧。

内核学习的心理问题

对于学习来说，无论是在学校的课堂学习，还是这里说的内核学习，效果好或者坏，最主要取决于两个方面——方法论和心理。注意，我无视了智商的差异，这玩意儿玄之又玄，岔开了说，属于迷信的范畴。

前面又是 Kernel 地图，又是如何入手，说的都是方法论的问题，那么这里要面对的就主要是心理上的问题。

而心理上的问题主要有两个，一个是盲目，就是在能够熟练适用 Linux 之前，对 Linux 为何物还说不出个道道来，就迫不及待的盲目的去研究内核的源代码。这一部分人会觉得既然是学习内核，那么耗费时间在熟悉 Linux 的基本操作上纯粹是浪费宝贵的时间和感情。不过这样虽然很有韩峰同志的热情和干劲儿，但明显走入了一种心理误区。重述 Linus 的那句话：要先会使用它。

第二个就是恐惧。人类进化这么多年，面对复杂的物体和事情还是总会有天生的惧怕感，体现在内核学习上面就是：那么庞大复杂的内核代码，让人面对起来该情何以堪啊！

有了这种恐惧无力感存在，心理上就会去排斥面对接触内核源码，宁愿去抱着情景分析，搜集各种各样五花八门的内核书籍放在那里屯着，看了又忘，忘了又看，也不大情愿去认真细致得浏览源码。

这个时候，我们在心理上是脆弱得，我们忘记了芙蓉姐姐，工行女之所以红起来，不是她们有多好，而是因为她们得心理足够坚强。是的，除了向韩局长学习态度，我们还要向涌现出来的无数个芙蓉姐姐和工行女学习坚强的心理。

有必要再强调一次，学习内核，就是学习内核的源代码，任何内核有关的书籍都是基于

内核，而又不高于内核的。内核源码本身就是最好的参考资料，其他任何经典或非经典的书最多只是起到个辅助作用，不能也不应该取代内核代码在我们学习过程中的主导地位。

内核学习的相关资源

“世界上最缺的不是金钱，而是资源。”当我在一份报纸上看到这句大大标题时，我的第一反应是——作者一定是个自然环保主义者，然后我在羞愧得反省自身的同时油然生出一股对这样的无产主义理想者无比崇敬的情绪来。

于是，我继续往下看，“因此在 XXX 还未正式面市之时，前来咨询的客户已经不少，这些有眼光的购房者明白，谁能在目前最好的购房机会下最大化地占有绝版资源，谁就掌控了未来财富流向。”（为了避免做广告的嫌疑，请允许我使用 XXX 代替该楼盘的名字。）顿时，我悟道了！

其实，韩峰同志已经在日记里告诉了我们资源的重要性，因此我们在学习韩峰同志严谨细致的态度同时，还要领悟他对资源的灵活运用。只有在以内核源码为中心，坚持各种学习资源的长期建设不动摇，才能达到韩局长那样的高度，俯视 Linux 内核世界里的人生百态。

注意，这个观点与前面所说的学习效果主要取决于方法论和心理两个方面并不矛盾，它们属于不同层次上的问题。

内核文档

内核代码中包含有大量的文档，这些文档对于学习理解内核有着不可估量的价值，记住，在任何时候，它们在我们心目中的地位都应该高于那些各式的内核参考书。下面是一些内核新人所应该阅读的文档。

README

这个文件首先简单介绍了 Linux 内核的背景，然后描述了如何配置和编译内核，最后还告诉我们出现问题时应该怎么办。

Documentation/Changes

这个文件给出了用来编译和使用内核所需要的最小软件包列表。

Documentation/CodingStyle

这个文件描述了内核首选的编码风格，所有代码都应该遵守里面定义的规范。

Documentation/SubmittingPatches

Documentation/SubmittingDrivers

Documentation/SubmitChecklist

这三个文件都是描述如何提交代码的，其中 SubmittingPatches 给出创建和提交补丁的过程，SubmittingDrivers 描述了如何将设备驱动提交给 2.4、2.6 等不同版本的内核树，SubmitChecklist 则描述了提交代码之前需要 check 自己的代码应该遵守的某些事项。

Documentation/stable_api_nonsense.txt

这个文件解释了为什么内核没有一个稳定的内部 API（到用户空间的接口——系统调用——是稳定的），它对于理解 Linux 的开发哲学至关重要，对于将开发平台从其他操作系统转移到 Linux 的开发者来说也很重要。

Documentation/stable_kernel_rules.txt

解释了稳定版内核（stable releases）发布的规则，以及如何将补丁提交给这些版本。

Documentation/SecurityBugs

内核开发者对安全性问题非常关注，如果你认为自己发现了这样的问题，可以根据这个文件中给出的联系方式提交 bug，以便能够尽可能快的解决这个问题。

Documentation/kernel-docs.txt

这个文件列举了很多内核相关的文档和书籍，里面不乏经典之作。

Documentation/applying-patches.txt

这个文件回答了如何为内核打补丁。

Documentation/bug-hunting

这个文件是有关寻找、提交、修正 bug 的。

Documentation/HOWTO

这个文件将指导你如何成为一名内核开发者，并且学会如何同内核开发社区合作。它尽可能不包括任何关于内核编程的技术细节，但会给你指引一条获得这些知识的正确途径。

经典书籍

待到山花烂漫时，还是那些经典在微笑。

有关内核的书籍可以用汗牛充栋来形容，不过只有一些经典的神作经住了考验。首先是 5 本久经考验的神作（个人概括为“2+1+2”，第一个 2 是指 2 本全面讲解内核的书，中间的 1 指 1 本讲解驱动开发的书，后面的 2 则指 2 本有关内核具体子系统的书，你是否想到了某某广告里三个人突然站起单臂齐举高呼“1 比 1 比 1”的场景？）。

《Linux 内核设计与实现》

简称 LKD，从入门开始，介绍了诸如进程管理、系统调用、中断和中断处理程序、内核同步、时间管理、内存管理、地址空间、调试技术等方面，内容比较浅显易懂，个人认为是内核新人首先必读的书籍。新人得有此书，足矣！

《深入理解 Linux 内核》

简称 ULK，相比于 LKD 的内容不够深入、覆盖面不广，ULK 要深入全面得多。

前面这两本，一本提纲挈领，一本全面深入。

《Linux 设备驱动程序》

简称 LDD，驱动开发者都要人手一本了。

《深入理解 Linux 虚拟内存管理》

简称 LVMM，是一本介绍 Linux 虚拟内存管理机制的书。如果你希望深入的研究 Linux 的内存管理子系统，仔细的研读这本书无疑是最好的选择。

《深入理解 LINUX 网络内幕》

一本讲解网络子系统实现的书，通过这本书，我们可以了解到 Linux 内核是如何实现复杂的网络功能的。

这 5 本书各有侧重，正如下面的图所展示的那样，恰好代表了个人一直主张的内核学习方法：首先通过 LKD 或 ULK 了解内核的设计实现特点，对内核有个整体全局的认识和理解，然后可分为两个岔路，如果从事驱动开发，则钻研 LDD，如果希望对内核不是泛泛而谈而是有更深入的理解，则可以选择一个自己感兴趣的子系统，仔细分析它的代码，不懂的地方就通过社区、邮件列表或者直接发 Email 给 maintainer 请教等途径弄懂，切勿得过且过，这样分析下来，对同步、中断等等内核的很多机制也同样会非常了解，俗话说的一通则百通就是这个道理。当然，如果你选择研究的是内存管理或者网络，则可以有上面的两本书可以学习，如果是其他子系统，可能就没有这么好的运气了。

内核社区

最近几年，社区网站非常的热火，不过此社区非彼社区。

Linux 最大的一个优势就是它有一个紧密团结了众多使用者和开发者的社区，它的目标就是提供尽善尽美的内核。内核社区的中心是内核邮件列表（Linux Kernel Mailing List，LKML），我们可以在 http://vger.kernel.org/vger-lists.html#linux-kernel 上面看到订阅这个邮件列表的细节。

内核邮件列表的流量很大，每天都有几百条消息，这里是大牛们的战场，小牛们的天堂，任何一个内核开发者都可以从中受益非浅。

除了 LKML，大多数子系统也有自己独立的邮件列表来协调各自的开发工作，比如 USB 子系统的邮件列表可以在 http://www.linux-usb.org/mailing.html 上面订阅。

其他网络资源

除了内核邮件列表，还有很多其他的论坛或网站值得我们经常关注。我们要知道，网络上不仅有兽兽和凤姐，也不仅有犀利哥和韩局长。

http://www.kernel.org/

可以通过这个网站上下载内核的源代码和补丁、跟踪内核bug等。

http://kerneltrap.org

Linux和BSD内核的技术新闻。如果没时间跟踪LKML，那么经常浏览kerneltrap是个好主意。

http://lwn.net/

Linux weekly news，创建于1997年底的一个Linux新闻站点。

http://zh-kernel.org/mailman/listinfo/linux-kernel

这是内核开发的中文邮件列表，里面活跃着很多内核开发领域的华人，比如Herbert Xu,、Mingming Cao、Bryan Wu等。

http://linux.chinaunix.net/

全球最大的Linux/Unix中文技术社区。

……

模块机制与“Hello World!”

有一种感动，叫泪流满面，有一种机制，叫模块机制。显然，这种模块机制给那些 Linux 的发烧友们带来了方便，因为模块机制意味着人们可以把庞大的 Linux 内核划分为许许多多个小的模块。对于编写设备驱动程序的开发者来说，从此以后他们可以编写设备驱动程序却不需要把她编译进内核，不用 reboot 机器，她只是一个模块，当你需要她的时候，你可以把她抱入怀中（insmod），当你不再需要她的时候，你可以把她一脚踢开（rmmod）。

于是，忽如一夜春风来，内核处处是模块。让我们从一个伟大的例子去认识模块。这就是传说中的"Hello World!"，这个梦幻般的名字我们看过无数次了，每一次她出现在眼前，就意味着我们开始接触一种新的计算机语言了。（某程序员对书法十分感兴趣，退休后决定在这方面有所建树。于是花重金购买了上等的文房四宝。一日，饭后突生雅兴，一番磨墨拟纸，并点上了上好的檀香，颇有王羲之风范，又具颜真卿气势，定神片刻，泼墨挥毫，郑重地写下一行字：hello world）

请看下面这段代码，她就是 Linux 下的一个最简单的模块。当你安装这个模块的时候，她会用她特有的语言向你表白：“Hello，world!”，而后来你卸载了这个模块，你无情抛弃了她，她很伤心，她很绝望，但她没有抱怨，她只是淡淡地说，“Goodbye，cruel world!”（再见，残酷的世界!）

```
/************ hello.c *********************/

#include <linux/init.h>  /* Needed for the macros */
#include <linux/module.h> /* Needed for all modules */
MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("fudan_abc");

```

```
static int __init hello_init(void)
{
        printk(KERN_ALERT "Hello, world!\n");
        return 0;
}

static void __exit hello_exit(void)
{
        printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

你需要使用 module_init()和 module_exit()，你可以称它们为函数，不过实际上它们是一些宏，你可以不用去知道她们背后的故事，只需要知道，在 Linux Kernel 2.6 的世界里，你写的任何一个模块都需要使用它们来初始化或退出，或者说注册以及后来的注销。

当你用 module_init()为一个模块注册了之后，在你使用 insmod 这个命令去安装的时候，module_init()注册的函数将会被执行。而当你用 rmmod 这个命令去卸载一个模块的时候，module_exit()注册的函数将会被执行。module_init()被称为驱动程序的初始化入口(driver initialization entry point)。

怎么样演示以上代码的运行呢？没错，你需要一个 Makefile。

```
# To build modules outside of the kernel tree, we run "make"
# in the kernel source tree; the Makefile these then includes this
# Makefile once again.
# This conditional selects whether we are being included from the
# kernel Makefile or not.
ifeq ($(KERNELRELEASE),)

    # Assume the source tree is where the running kernel was built
    # You should set KERNELDIR in the environment if it's elsewhere
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    # The current directory is passed to sub-makes as argument
    PWD := $(shell pwd)

modules:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

modules_install:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install

clean:
        rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
```

```

.PHONY: modules modules_install clean

else
    # called from kernel build system: just declare what our modules are
    obj-m := hello.o
endif
```

在 lwn.net 上可以找到这个例子，你可以把以上两个文件放在你的某个目录下，然后执行 make，也许你不一定能成功，因为 Linux Kernel 2.6 要求你编译模块之前，必须先在内核源代码目录下执行 make，换言之，你必须先配置过内核，执行过 make，然后才能 make 你自己的模块。原因就不细说了，你按着要求的这么去做就行了。在内核顶层目录 make 过之后，你就可以在你当前放置 Makefile 的目录下执行 make 了。make 之后你就应该看到一个叫做 hello.ko 的文件生成了，恭喜你，这就是你将要测试的模块。

执行命令，

```
#insmod hello.ko
```

同时在另一个窗口，用命令 tail -f /var/log/messages 察看日志文件，你会看到 Hello world 被打印了出来。再执行命令，

```
#rmmod hello.ko
```

此时，在另一窗口你会看到 Goodbye，cruel world！被打印了出来。

到这里，我该恭喜你，因为你已经能够编写 Linux 内核模块了。这种感觉很美妙，不是吗？你可以嘲笑秦皇汉武略输文采唐宗宋祖稍逊风骚，还可以嘲笑一代天骄成吉思汗只识弯弓射大雕了。是的，阿娇姐姐告诉我们，只要我喜欢，还有什么不可以。

日后我们会看到，2.6 内核中，每个模块都是以 module_init 开始，以 module_exit 结束。对大多数人来说没有必要知道这是为什么，记住就可以了，对大多数人来说，这就像是 1+1 为什么等于 2 一样，就像是两点之间最短的是直线，不需要证明，如果一定要证明两点之间直线最短，可以扔一块骨头在 B 点，让一条狗从 A 点出发，你会发现狗走的是直线，是的，狗都知道，咱还能不知道吗？

设备模型（上）

对于驱动开发来说，设备模型的理解是根本，毫不夸张得说，理解了设备模型，再去看那些五花八门的驱动程序，你会发现自己站在了另一个高度，从而有了一种俯视的感觉，就像凤姐俯视知音和故事会，韩峰同志俯视女下属。

顾名而思义就知道设备模型是关于设备的模型，既不是任小强们的房模，也不是张导的炮模。对咱们写驱动的和不写驱动的人来说，设备的概念就是总线和与其相连的各种设备了。电脑城的 IT 工作者都会知道设备是通过总线连到计算机上的，而且还需要对应的驱动才能用，可是总线是如何发现设备的，设备又是如何和驱动对应起来的，它们经过怎样的艰辛才找到命里注定的那个他，它们的关系如何，白头偕老型的还是朝三暮四型的，这些问题就不

是他们关心的了，而是咱们需要关心的。在房市股市千锤百炼的咱们还能够惊喜的发现，这些疑问的中心思想中心词汇就是总线、设备和驱动，没错，它们就是咱们这里要聊的 Linux 设备模型的名角。

总线、设备、驱动，也就是 bus、device、driver，既然是名角，在内核里都会有它们自己专属的结构，在 include/linux/device.h 里定义。

```
struct bus_type {
        const char              * name;
        struct module           * owner;

        struct kset             subsys;
        struct kset             drivers;
        struct kset             devices;
        struct klist            klist_devices;
        struct klist            klist_drivers;

        struct blocking_notifier_head bus_notifier;

        struct bus_attribute    * bus_attrs;
        struct device_attribute * dev_attrs;
        struct driver_attribute * drv_attrs;
        struct bus_attribute drivers_autoprobe_attr;
        struct bus_attribute drivers_probe_attr;

        int (*match)(struct device * dev, struct device_driver * drv);
        int (*uevent)(struct device *dev, char **envp,
        int num_envp, char *buffer, int buffer_size);
        int             (*probe)(struct device * dev);
        int             (*remove)(struct device * dev);
        void            (*shutdown)(struct device * dev);

        int (*suspend)(struct device * dev, pm_message_t state);
        int (*suspend_late)(struct device * dev, pm_message_t state);
        int (*resume_early)(struct device * dev);
        int (*resume)(struct device * dev);

        unsigned int drivers_autoprobe:1;
};

struct device_driver {
        const char              * name;
        struct bus_type         * bus;

        struct kobject          kobj;
        struct klist            klist_devices;
```

```
    struct klist_node      knode_bus;

    struct module          * owner;
    const char            * mod_name;  /* used for built-in modules */
    struct module_kobject   * mkobj;

    int     (*probe)       (struct device * dev);
    int     (*remove)      (struct device * dev);
    void    (*shutdown)     (struct device * dev);
    int     (*suspend)      (struct device * dev, pm_message_t state);
    int     (*resume)      (struct device * dev);
};

struct device {
    struct klist       klist_children;
    struct klist_node   knode_parent;       /* node in sibling list */
    struct klist_node   knode_driver;
    struct klist_node   knode_bus;
    struct device      *parent;

    struct kobject kobj;
    char    bus_id[BUS_ID_SIZE];    /* position on parent bus */
    struct device_type  *type;
    unsigned       is_registered:1;
    unsigned       uevent_suppress:1;

     struct semaphore   sem;    /* semaphore to synchronize calls to
                      * its driver.
                      */

    struct bus_type * bus;      /* type of bus device is on */
    struct device_driver *driver;  /* which driver has allocated this
                        device */
    void        *driver_data;   /* data private to the driver */
    void        *platform_data; /* Platform specific data, device
                        core doesn't touch it */
    struct dev_pm_info  power;

#ifdef CONFIG_NUMA
    int     numa_node;  /* NUMA node this device is close to */
#endif
    u64     *dma_mask;  /* dma mask (if dma'able device) */
    u64     coherent_dma_mask;/* Like dma_mask, but for
                         alloc_coherent mappings as
```

```
                            not all hardware supports
                            64 bit addresses for consistent
                            allocations such descriptors. */

        struct list_head    dma_pools;  /* dma pools (if dma'ble) */

        struct dma_coherent_mem *dma_mem; /* internal for coherent mem
                            override */
        /* arch specific additions */
        struct dev_archdata archdata;

        spinlock_t      devres_lock;
        struct list_head    devres_head;

        /* class_device migration path */
        struct list_head    node;
        struct class        *class;
        dev_t           devt;       /* dev_t, creates the sysfs "dev" */
        struct attribute_group  **groups;   /* optional groups */

        void    (*release)(struct device * dev);
};
```

有没有发现它们的共性是什么？对，不是很傻很天真，而是很长很复杂。不过不妨把它们看成艺术品，既然是艺术，当然不会让你那么容易的就看懂了，不然怎么称大师称名家。这么想想咱们就会比较的宽慰了，阿 Q 是鲁迅对咱们 80 后最大的贡献。

我知道进入了 21 世纪，最缺的就是耐性，房价股价都让咱们没有耐性，内核的代码也让人没有耐性。不过做为最没有耐性的一代人，还是要平心静气的扫一下上面的结构，我们会发现，struct bus_type 中有成员 struct kset drivers 和 struct kset devices，同时 struct device 中有两个成员 struct bus_type * bus 和 struct device_driver *driver，struct device_driver 中有两个成员 struct bus_type * bus 和 struct klist klist_devices。先不说什么是 klist、kset，光从成员的名字看，它们就是一个完美的三角关系。我们每个人心中是不是都有两个她？一个梦中的她，一个现实中的她。

凭一个男人的直觉，我们可以知道，struct device 中的 bus 表示这个设备连到哪个总线上，driver 表示这个设备的驱动是什么，struct device_driver 中的 bus 表示这个驱动属于哪个总线，klist_devices 表示这个驱动都支持哪些设备，因为这里 device 是复数，又是 list，更因为一个驱动可以支持多个设备，而一个设备只能绑定一个驱动。当然，struct bus_type 中的 drivers 和 devices 分别表示了这个总线拥有哪些设备和哪些驱动。

单凭直觉，张钰红不了。我们还需要看看什么是 klist、kset。还有上面 device 和 driver 结构里出现的 kobject 结构是什么？作为一个五星红旗下长大的孩子，我可以肯定的告诉你，kobject 和 kset 都是 Linux 设备模型中最基本的元素，总线、设备、驱动是西瓜，kobjcet、klist 是种瓜的人，没有幕后种瓜人的汗水不会有清爽解渴的西瓜，我们不能光知道西瓜的的甜，还要知道种瓜人的辛苦。kobject 和 kset 不会在意自己的得失，它们存在的意义在于把

总线、设备和驱动这样的对象连接到设备模型上。种瓜的人也不会在意自己的汗水，在意的只是能不能送出甜蜜的西瓜。

一般来说应该这么理解，整个 Linux 的设备模型是一个 OO 的体系结构，总线、设备和驱动都是其中鲜活存在的对象，kobject 是它们的基类，所实现的只是一些公共的接口，kset 是同种类型 kobject 对象的集合，也可以说是对象的容器。只是因为 C 里不可能会有 C++里类的 class 继承、组合等的概念，只有通过 kobject 嵌入到对象结构里来实现。这样，内核使用 kobject 将各个对象连接起来组成了一个分层的结构体系，就好像马列主义将我们 13 亿人也连接成了一个分层的社会体系一样。kobject 结构里包含了 parent 成员，指向了另一个 kobject 结构，也就是这个分层结构的上一层结点。而 kset 是通过链表来实现的，这样就可以明白，struct bus_type 结构中的成员 drivers 和 devices 表示了一条总线拥有两条链表，一条是设备链表，一条是驱动链表。我们知道了总线对应的数据结构，就可以找到这条总线关联了多少设备，又有哪些驱动来支持这类设备。

那么 klist 呢？其实它就包含了一个链表和一个自旋锁，我们暂且把它看成链表也无妨，本来在 2.6.11 内核里，struct device_driver 结构的 devices 成员就是一个链表类型。这么一说，咱们上面的直觉都是正确的，如果买股票，摸彩票时直觉都这么管用，就不会有咱们这被压扁的一代了。

现在的人都知道，三角关系很难处。那么总线、设备和驱动之间是如何和谐共处那？先说说总线中的那两条链表是怎么形成的。内核要求每次出现一个设备就要向总线汇报，或者说注册，每次出现一个驱动，也要向总线汇报，或者说注册。比如系统初始化的时候，会扫描连接了哪些设备，并为每一个设备建立起一个 struct device 的变量，每一次有一个驱动程序，就要准备一个 struct device_driver 结构的变量。把这些变量统统加入相应的链表，device 插入 devices 链表，driver 插入 drivers 链表。这样通过总线就能找到每一个设备，每一个驱动。然而，假如计算机里只有设备却没有对应的驱动，那么设备无法工作。反过来，倘若只有驱动却没有设备，驱动也起不了任何作用。在他们遇见彼此之前，双方都如同路埂的野草，一个飘啊飘，一个摇啊摇，谁也不知道未来在哪里，只能在生命的风里飘摇。于是总线上的两张表里就慢慢的就挂上了那许多孤单的灵魂。devices 开始多了，drivers 开始多了，他们像是两个来自世界，devices 们彼此取暖，drivers 们一起狂欢，但他们有一点是相同的，都只是在等待属于自己的那个另一半。

现在，总线上的两条链表已经有了，这个三角关系三个边已经有了两个，剩下的那个那？链表里的设备和驱动又是如何联系那？先有设备还是先有驱动？很久很久以前，在那激情燃烧的岁月里，先有的是设备，每一个要用的设备在计算机启动之前就已经插好了，插放在它应该在的位置上，然后计算机启动，然后操作系统开始初始化，总线开始扫描设备，每找到一个设备，就为其申请一个 struct device 结构，并且挂入总线中的 devices 链表中来，然后每一个驱动程序开始初始化，开始注册其 struct device_driver 结构，然后它去总线的 devices 链表中去寻找(遍历)，去寻找每一个还没有绑定驱动的设备，即 struct device 中的 struct device_driver 指针仍为空的设备，然后它会去观察这种设备的特征，看是否是他所支持的设备，如果是，那么调用一个叫做 device_bind_driver 的函数，然后他们就结为了秦晋之好。换句话说，把 struct device 中的 struct device_driver driver 指向这个驱动，而 struct device_driver driver 把 struct device 加入他的那张 struct klist klist_devices 链表中来。就这样，bus、device 和 driver，这三者之间或者说他们中的两两之间，就给联系上了。知道其中之一，就能找到另外两个。一荣俱荣，一损俱损。

但现在情况变了，在这红莲绽放的日子里，在这樱花伤逝的日子里，出现了一种新的名

词，叫热插拔。设备可以在计算机启动以后在插入或者拔出计算机了。因此，很难再说是先有设备还是先有驱动了。因为都有可能。设备可以在任何时刻出现，而驱动也可以在任何时刻被加载，所以，出现的情况就是，每当一个 struct device 诞生，它就会去 bus 的 drivers 链表中寻找自己的另一半，反之，每当一个一个 struct device_driver 诞生，它就去 bus 的 devices 链表中寻找它的那些设备。如果找到了合适的，那么 OK，和之前那种情况一下，调用 device_bind_driver 绑定好。如果找不到，没有关系，等待吧，等到昙花再开，等到风景看透，心中相信，这世界上总有一个人是你所等的，只是还没有遇到而已。

设备模型（下）

设备模型拍得再玄幻，它也只是个模型，必须得落实在具体的子系统，否则就只能抱着个最佳技术奖空遗恨。既然前面已经以 USB 子系统的实现分析示例了分析内核源码应该如何入手，那么这里就仍然以 USB 子系统为例，看看设备模型是如何软着陆的。

内核中 USB 子系统的结构

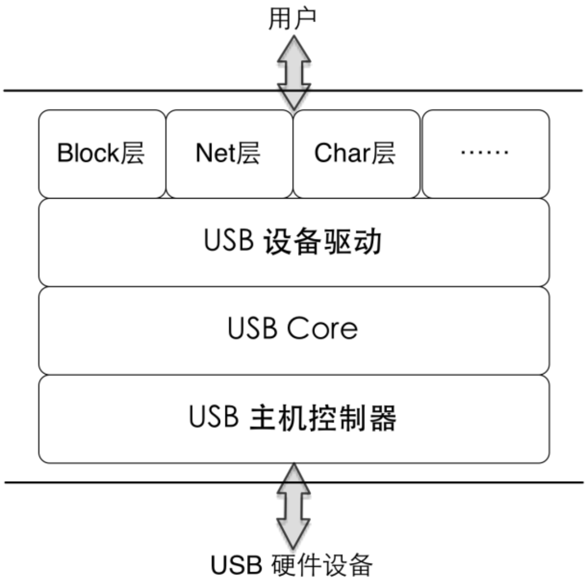
我们已经知道了 USB 子系统的代码都位于 drivers/usb 目录下面，也认识了一个很重要的目录——core 子目录。现在，我们再来看一个很重要的模块——usbcore。你可以使用“lsmod”命令看一下，在显示的结果里能够找到有一个模块叫做 usbcore。

```
localhost:/usr/src/linux-2.6.23/drivers/usb/core # lsmod
Module                  Size  Used by
af_packet              55820  2
raw                    89504  0
nfs                   230840  2
lockd                  87536  2 nfs
nfs_acl                20352  1 nfs
sunrpc                172360  4 nfs,lockd,nfs_acl
ipv6                  329728  36
button                 24224  0
battery                27272  0
ac                     22152  0
apparmor               73760  0
aamatch_pcre           30720  1 apparmor
loop                   32784  0
usbhid                 60832  0
dm_mod                 77232  0
ide_cd                 57120  0
hw_random              22440  0
ehci_hcd               47624  0
cdrom                  52392  1 ide_cd
uhci_hcd               48544  0
shpchp                 61984  0
bnx2                  157296  0
usbcore               149288  4 usbhid,ehci_hcd,uhci_hcd
```

```
e1000                  130872  0
pci_hotplug              44800  1 shpchp
reiserfs                239616  2
edd                     26760  0
fan                     21896  0
……
```

找到了 usbcore 那一行吗？core 就是核心，基本上你要在你的电脑里用 USB 设备，那么两个模块是必须的：一个是 usbcore，这就是核心模块；另一个是主机控制器的驱动程序，比如这里 usbcore 那一行我们看到的 ehci_hcd 和 uhci_hcd，你的 USB 设备要工作，合适的 USB 主机控制器模块也是必不可少的。

usbcore 负责实现一些核心的功能，为别的设备驱动程序提供服务，提供一个用于访问和控制 USB 硬件的接口，而不用去考虑系统当前存在哪种主机控制器。至于 core、主机控制器和 USB 驱动三者之间的关系，如下图所示。



USB 驱动和主机控制器就像 core 的两个保镖，协议里也说了，主机控制器的驱动（HCD）必须位于 USB 软件的最下一层。HCD 提供主机控制器硬件的抽象，隐藏硬件的细节，在主机控制器之下是物理的 USB 及所有与之连接的 USB 设备。而 HCD 只有一个客户，对一个人负责，就是 usbcore。usbcore 将用户的请求映射到相关的 HCD，用户不能直接访问 HCD。

core 为咱们完成了大部分的工作，因此咱们写 USB 驱动的时候，只能调用 core 的接口，core 会将咱们的请求发送给相应的 HCD。

USB 子系统与设备模型

关于设备模型，最主要的问题就是，bus、device、driver 是如何建立联系的？换言之，这三个数据结构中的指针是如何被赋值的？绝对不可能发生的事情是，一旦为一条总线申请了一个 struct bus_type 的数据结构之后，它就知道它的 devices 链表和 drivers 链表会包含哪些东西，这些东西一定不会是先天就有的，只能是后天填进来的。

具体到 USB 子系统，完成这个工作的就是 USB core。USB core 的代码会进行整个 USB 系统的初始化，比如申请 struct bus_type usb_bus_type，然后会扫描 USB 总线，看线上连接

了哪些 USB 设备，或者说 Root Hub 上连了哪些 USB 设备，比如说连了一个 USB 键盘，那么就为它准备一个 struct device，根据它的实际情况，为这个 struct device 赋值，并插入 devices 链表中来。

又比如 Root Hub 上连了一个普通的 Hub，那么除了要为这个 Hub 本身准备一个 struct device 以外，还得继续扫描看这个 Hub 上是否又连了别的设备，有的话继续重复之前的事情，这样一直进行下去，直到完成整个扫描，最终就把 usb_bus_type 中的 devices 链表给建立了起来。

那么 drivers 链表呢？这个就不用 bus 方面主动了，而该由每一个 driver 本身去 bus 上面登记，或者说挂牌。具体到 USB 子系统，每一个 USB 设备的驱动程序都会对应一个 struct usb_driver 结构，其中有一个 struct device_driver driver 成员，USB core 为每一个设备驱动准备了一个函数，让它把自己的这个 struct device_driver driver 插入到 usb_bus_type 中的 drivers 链表中去。而这个函数正是我们此前看到的 usb_register。而与之对应的 usb_deregister 所从事的正是与之相反的工作，把这个结构体从 drivers 链表中删除。

而 struct bus_type 结构的 match 函数在 USB 子系统里就是 usb_device_match 函数，它充当了一个红娘的角色，在 USB 总线的 USB 设备和 USB 驱动之间牵线搭桥，类似于交大 BBS 上的鹊桥版，虽然它们上面的条件都琳琅满目的，但明显这里 match 的条件不是那么的苛刻，要更为实际些。

可以说，USB core 的确是用心良苦，为每一个 USB 设备驱动做足了功课，正因为如此，作为一个实际的 USB 设备驱动，它在初始化阶段所要做的事情就很少，很简单了，直接调用 usb_register 即可。事实上，没有人是理所当然应该为你做什么的，但 USB core 这么做了。所以每一个写 USB 设备驱动的人应该铭记，USB 设备驱动绝不是一个人在工作，在他身后，是 USB core 所提供的默默无闻又不可或缺的支持。

驱动开发三件宝：spec、datasheet 与内核源码

设备模型之外，对于驱动程序的开发者来说，有三样东西是不可缺少的：第一是协议或标准的 spec，也就是规范，比如 usb 协议规范；第二是硬件的 datasheet，即你的驱动要支持的硬件的手册；第三就是内核里类似驱动的源代码，比如你要写触摸屏驱动的话，就可以参考内核里已经有的一些触摸屏驱动。

spec、datasheet、内核源代码这三样东西对于每个开发设备驱动的人来说都是再寻常不过了，但正是因为它们的普通，所以在很多人眼里都被归为被忽视的群体。于是大家开发驱动的过程中，遇到问题的时候首先想到的可能还是“问问牛人怎么解决吧”、“旁边要是有个牛人该多好”，因为牛人的稀有，所以知道牛人的价值，而又因为 spec、datasheet 和内核源代码的唾手可得，所以常常体会不到它们在解决问题时的重要性。

当然我并不是贬低牛人的价值，宣扬依赖牛人不好，如果你很幸运身边真就有牛人这种稀缺资源的话，自然是要好好利用，也可以少走很多弯路，节省很多摸索的时间。只是人生不如意十之八九，多数人还是没有这份幸运的，所以与其遍寻牛人讨教，不如多依赖依赖自己，多利用利用自己身边有的资源去寻找解决问题的途径。

对这三样看似普通的东西，关键在于很好的去利用，而不是拥有。就说 USB 吧，USB

驱动和 USB 设备如何进行交流，交流的方式，交流过程中出现了什么问题是什么引起的等等都在 USB spec 里有描述，而你的 USB 设备支持多少种配置包含多少端点只有设备的 datasheet 才知道。协议的 spec 和设备的 datasheet 是最好的参考资料，驱动开发调试中出现的问题绝大部分都能在它们的某个角落里找到答案。而内核中类似设备的驱动源代码是最好的模版，对很多硬件设备，你都可以在内核找到同种设备的驱动代码进行参考实现，甚至于可以拷贝或共享大部分的代码，只进行局部的修改，比如说位于 drivers/input/touchscreen 目录下的各个触摸屏驱动，它们之间的代码很多都是类似的甚至是相同的。

如果你不仅仅只是打算写驱动，而是还想阅读内核中实现某种总线、设备的源代码，钻研它们的实现机制，那协议的 spec 就犹为重要，它们在代码里的体现无处不在，你需要在阅读代码前就对协议规范有个整体的理解。形象点说，spec 是理论基础，内核代码是具体实现，理论懂了，看代码就和看故事会差不多了。

Linux 内核问题门——学习问题、经验集锦

陈宪章说："学贵有疑，小疑则小进，大疑则大进。疑者，觉悟之机也，一番觉悟一番长进。"培根说："多问的人将多得。"还在学校的时候导师在激情讲演之后对着会议室里形态各异但均静默不语的我们痛心疾首的说："会提问题很重要啊，同志们！不会提问题怎么有资格做研究！"

这样铿锵有力的训诫今日想起仍觉深受刺激，于是就要不可避免得要做出一些反应来。不过一是因为咱这年代还没有非主流的说法，二是因为也没有冯仰妍同学的性别优势，不可能受到刺激就整出个门来。咱能够做到的最大反应也就是在这里开贴专门探讨探讨内核学习的相关问题，为了稍微增加那么一些广告效应，就称为"问题门"吧。

使用"问题门"的称呼，一是内心里潜藏的那点低级趣味想去沾点近些年层出不穷各种各样的"门"的仙气，二是在内核的学习过程中的确实实在在的存在着这样的一个"门"，横亘在我们的面前，跨过去便海阔天空是另一番世界，但却是让无数人竞折腰，百思不得其钥匙。

另外，这个"问题门"也算是为拙作《Linux 内核修炼之道》制作的一个小插曲，以感谢精华篇发布过程中很多朋友的关心与支持，希望通过对大家内核学习过程中遇到的问题与经验心得做一番展示，来帮助还在门外的朋友寻找到这扇门的钥匙。

我先整理一些在与很多网友交流过程中遇到的部分内核学习问题和自己的一些经验，来作为这个"问题门"的雏形，大家也可以在评论里提出自己的问题和分享自己的学习心得，我会及时地对其进行整理汇总，大家一起将"问题门"逐步完善，帮助后来者和有需要地人不再为"入门"而苦恼。希望这是一篇能够成长得文章！

Linux 内核学习常见问题

2010 年 3 月 24 日更新

"问题门"第 5 回：学习 Linux 内核，应该从 Linux 哪个版本代码开始阅读更好呢？

fudan_abc 的回答：

个人建议从新的内核开始，固然新内核的代码非常庞大，但并没有说非要求大求全，追

求每个部分都要理解。

学内核忌讳求大而全，如果对哪部分比较感兴趣，研究相关的源码和 change 就行了，当然仁者见仁智者见智，自己如果觉得从低版本开始更好更适合，那采用这种方式也未尝不可，毕竟各人的路还是各自走的。

2010 年 3 月 23 日更新

“问题门”第 3 回：通常，语言及其库的学习分为几个层次，1.熟练使用，2.阅读源码，了解实现原理，3.对源码进行扩展。那么 linux kernel 怎么划分层次，每个层次如何达到？（hust_tulip 提出）

fudan_abc 的回答：

问题中的三个层次对应到 linux 内核的学习上：“熟练使用”就是要能够熟练的使用 linux 系统；“阅读源码”就是指“学习内核就是学习内核源代码”，必须勇敢的去学习内核源码；“对源码进行扩展”可以对应于融入内核社区，参与内核的开发。

这也正好在一定程度上锲合了本书前言里对内核学习划分的几个层次：全面了解抓基本，兴趣导向深钻研；融入社区做贡献，坚持坚持再坚持。

——详见修炼之道 之 前言

“问题门”第 4 回：每个层次的学习都有什么对应的参考资料以及网络资源？（hust_tulip 提出）

fudan_abc 的回答：

首先是“全面了解抓基本”，这个层次，最好的书自然就是 lkd 和 ulk 了，这两本书，一本提纲挈领，一本全面深入，都能很好的帮助全面的理解内核的整体机制。新人的话，一本 lkd 就足亦了。

第二个层次“兴趣导向深钻研”，这个层次就是要以内核源码为中心，选择内核中一个自己感兴趣的部分，以韩峰同志对待日记的态度，严谨而细致的仔细分析它的代码，不懂的地方就通过社区、邮件列表或者直接发 Email 给 maintainer 请教等途径弄懂，切勿得过且过。至于这个层次的参考书么，网络子系统的有《深入理解 LINUX 网络内幕》，内存管理的有《深入理解 Linux 虚拟内存管理》，推荐看英文版，呵呵，usb 的可以看我们的《Linux 那些事儿》，其它子系统的还没注意到有什么专门讲解的，不过内核源码本身就是最好的参考资料了。

至于第三个层次“融入社区做贡献”，就是要努力融入到内核的开发社区，经过前两个层次的修炼，此时你已经不会再是社区中潜水小白的角色，而是会针对某个问题发表自己的见解。可以尝试参与到内核的开发中去。相关资源有很多，详细可以参考修炼之道精华篇的（9）内核学习资源

最后一层就是坚持了，不管遇到什么挫折都不放弃，就像咱们的袁教授不管遭受到什么样的辱骂都要坚持不断的发疯一样，有这样的精神，何愁在 linux 内核的学习道路上修不成大道那？

2010 年 3 月 22 日更新

"问题门"第 1 回：我是一个初学者，两眼一抹黑，我该如何学习内核？

fudan_abc 的回答：

这个问题每个初学者都无法回避，它非常之大，完全可以做为整个"问题门"的框架而存在，其他的各种问题都不过是在这个框架上装饰和完善。

同时这个问题并没有一个标准的答案，只有一些学习的脉络可以遵循，祝早日"入门"。

第一步：先会使用它。连 Linux 是什么、基本操作都不会就去研究内核，纯属扯淡，"门"都没有。

第二步：看懂内核源码需要一些操作系统、C 语言等的基础。

第三步：找本合适的内核参考书，让它帮助你对内核有个整体的理解和认识，

第四步：要能够动手配置编译内核，还要基本看得懂内核中的 Kconfig 和 Makefile 文件。

最后，记住："学习内核，就是学习内核的源代码，任何内核有关的书籍都是基于内核，而又不高于内核的。内核源码本身就是最好的参考资料，其他任何经典或非经典的书最多只是起到个辅助作用，不能也不应该取代内核代码在我们学习过程中的主导地位。"

因此你要做得是选择内核的一个部分或子系统，以韩峰同志对待日记的态度，严谨而细致得理解每一段代码的实现，多问多想多记。切勿抱着走马观花，得过且过的态度。

"问题门"第 2 回：学习内核需要什么基础知识？

albcamus 的回答：

（1）需要掌握操作系统理论的最初级的知识。

不需要通读并理解《操作系统概念》《现代操作系统》等巨著，但总要知道分时（time-shared）和实时（real-time）的区别是什么，进程是个什么东西，CPU 和系统总线、内存的关系（很粗略即可），等等。

（2）熟练使用 C 语言。

不需要已经很精通 C 语言，只要能熟练编写 C 程序，能看懂链表、散列表等数据结构的 C 实现，用过 gcc 编译器，就可以了。当然，如果已经精通 C 语言显然是大占便宜的。

（3）了解 CPU 的相关知识。

Linux 内核学习经验

1. 内核学习的心理误区

心理上的问题主要有两个，一个是盲目，就是在能够熟练使用 Linux 之前，对 Linux 为何物还说不出个道道来，就迫不及待的盲目的去研究内核的源代码。重述 Linus 的那句话：要先会使用它。

第二个就是恐惧。人类进化这么多年，面对复杂的物体和事情还是总会有天生的惧怕感，体现在内核学习上面就是：那么庞大复杂的内核代码，让人面对起来该情何以堪啊！

有了这种恐惧无力感存在，心理上就会去排斥面对接触内核源码，宁愿去抱着情景分析，搜集各种各样五花八门的内核书籍放在那里屯着，看了又忘，忘了又看，也不大情愿去认真细致得浏览源码。

——详见修炼之道精华篇（9）内核学习的心理问题

2. 学习内核就是学习内核的源代码

学习内核，就是学习内核的源代码，任何内核有关的书籍都是基于内核，而又不高于内核的。内核源码本身就是最好的参考资料，其他任何经典或非经典的书最多只是起到个辅助作用，不能也不应该取代内核代码在我们学习过程中的主导地位。

3. 要抱着严谨细致的态度分析内核源码

既然要学习内核源码，就要经常对内核代码进行分析，而内核代码千千万，还前仆后继的不断往里加，这就让大部分人都有种雾里看花花不见的无助感。不过不要怕，孔老夫子早就留给我们了应对之策：敏于事而慎于言，就有道而正焉，可谓好学也已。这就是说，做事要踏实才是好学生好同志，要遵循严谨的态度，去理解每一段代码的实现，多问多想多记。如果抱着走马观花，得过且过的态度，结果极有可能就是一边看一边丢，没有多大的收获。

只要你使用这样的态度开始分析内核，那么无论你选择内核的哪个部分作为切入点，比如 USB，比如进程管理，在花费相对不算很多的时间之后，你就会发现你对内核的理解会上升到另外一个高度，一个抱着情景分析，抱着 0.1 内核完全注释，抱着各种各样的内核书籍翻来覆去的看很多遍又忘很多遍都无法达到的高度。

——详见修炼之道精华篇（6）与精华篇（7）分析内核源码如何入手？

4. 通过 Kconfig 与 Makefile 定位目标代码

毫不夸张地说，Kconfig 和 Makefile 是我们浏览内核代码时最为依仗的两个文件。基本上，Linux 内核中每一个目录下边都会有一个 Kconfig 文件和一个 Makefile 文件。 对于一个希望能够在 Linux 内核的汪洋代码里看到一丝曙光的人来说，将它们放在怎么重要的地位都不过分。

Kconfig 和 Makefile 就是 Linux Kernel 迷宫里的地图。地图引导我们去认识一个城市，而 Kconfig 和 Makefile 则可以让我们了解一个 Kernel 目录下面的结构。我们每次浏览 kernel 寻找属于自己的那一段代码时，都应该首先看看目录下的这两个文件。就像利用地图寻找目的地一样，我们需要利用 Kconfig 和 Makefile 来寻找所要研究的目标代码。

——详见修炼之道精华篇（5）Kernel 地图：Kconfig 与 Makefile